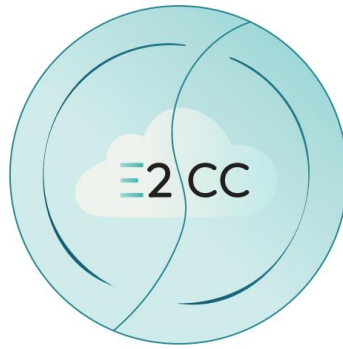




D2.2 Monitoring systems for fault and security incident detection in Cloud-Edge environments – Phase 1

Propriétés du document

Numéro de contrat	DOS0235220/00 – DOS0235222/00
Niveau de dissémination	<Public>
Nature	REPORT
Auteurs	Damien Monguillon, Thomas Ropars, Cheickna Sylla, Léonard Pain-teaux, Pascal Berthou, Philippe Owezarski
Date	26 Juin 2026
Etat	Final



Ce projet a été financé par l'Etat dans le cadre de France 2030.

Historique des changements

Version	Date	Auteur, Organisation	Description des Changements
0.1	11/12/2025	Damien Monguillon, NBS	Initialisation du document
0.2	02/06/2026	Thomas Ropars, UGA	Ajout des contributions UGA
0.3	05/06/2026	Cheickna Sylla, Léonard Painteaux, Pascal Berthou, Philippe Owezarski	LAAS contribution
1.0	26/06/2026	Philippe Owezarski, Damien Monguillon	Relecture et ajustements finaux

Table des matières

1	INTRODUCTION.....	5
2	OBJECTIVES	6
3	TECHNICAL CONTEXT.....	7
4	RESTITUTION OF CONTRIBUTORS' CONTRIBUTIONS.....	9
4.1	CNRS-LAAS.....	9
4.1.1	<i>Introduction.....</i>	<i>9</i>
4.1.2	<i>Challenges.....</i>	<i>10</i>
4.1.3	<i>Work Carried out.....</i>	<i>12</i>
4.1.4	<i>Security.....</i>	<i>52</i>
4.1.5	<i>Conclusion.....</i>	<i>53</i>
4.2	UGA.....	58
4.2.1	<i>Introduction.....</i>	<i>58</i>
4.2.2	<i>Challenges.....</i>	<i>58</i>
4.2.3	<i>Work Carried out.....</i>	<i>59</i>
5	CONCLUSION AND NEXT STEPS.....	63
6	ACRONYMS AND ABBREVIATIONS.....	64

Liste des tableaux

TABLE 1 : Acronyms and Abbreviations.....	64
---	----

Liste des figures

Figure 1 :High-level diagram of the Cloud-Edge Orchestration Solutions.....	8
Figure 2 : OVERALL ARCHITECTURE OF THE MONITORING system implemented.....	13
Figure 3 : GANTT CHART REPRESENTATION OF a distributed trace (Jaeger interface).....	14
Figure 4 : LIST OF RESOURCES PROVIDED BY SLICES-RI.....	15
Figure 5 : STATUS OF CURRENTLY RUNNING VM 1.....	15
Figure 6 : KUBERNETES NODE CONFIGURATION in the host.yaml inventory file.....	16
Figure 7 : cluster.yml configuration file provided by Kubespray [Kubespray].....	17
Figure 8 : Kubernetes Cluster.....	18
Figure 9 : Operational architecture of the otel-demo application [OTEL-DEMO].....	19
Figure 10 : Excerpt from the load generation script locustfile.py provided by otel-demo [Locustfile].....	20
Figure 11 : Summary excerpt from the otel-collector-agent ConfigMap with the observability pipeline implementation (Receivers, Processors, Exporters) [Prometheus].	22
Figure 12 : Excerpt from the grafana-datasources ConfigMap [Grafana Datasources].....	23
Figure 13 : Result of executing the Helm command, confirming that the cluster has processed the deployment of the opentelemetry-demo chart.....	24
Figure 14 : List and status of Pods in the otel-demo namespace.....	25
Figure 15 : Distribution of microservices across the cluster's various nodes, illustrating the load-balancing mechanism implemented by the kube-scheduler.....	26
Figure 16 : Local access to the application's web interfaces, Jaeger, and Grafana.....	29
Figure 17 : Excerpt from the add_aws.yaml manifest configuring the awss3 exporter and adding it to the metrics and traces pipelines.....	30
Figure 18 : MinIO database web interface.....	31
Figure 19 : Overall architecture and Chaos Mesh injection mechanisms [Chaos Mesh Arch]...33	
Figure 20 : Process for creating an authentication token.....	35
Figure 21 : Authentication token creation process.....	37

Figure 22 : Jaeger interface: home screen, Gantt chart, payment span tags, and the Process section.....	39
Figure 23 : Screenshot of the overall analysis and a detailed trace of the payment service.....	42
Figure 24 : How the Python code works.....	43
Figure 25 : Configuring and executing the network attack with a 2-second latency injection.....	47
Figure 26 : Execution of the NetworkChaos attack.....	47
Figure 27 : Result of the Chaos-Mesh execution.....	49
Figure 28 : StressChaos Configuration and Execution.....	50
Figure 29 : Capture of the cartservice Jaeger trace at 1.47 ms and the state of node4.....	51
Figure 30 : Impact of the characteristics of anomaly detectors on the performance of a cloud service.....	60
Figure 31 : Ctl-SRNN: How it works.....	61

1 Introduction

Deliverable 2.2 concerns the development and implementation of advanced monitoring systems to detect failures and security incidents in Cloud-Edge environments. This deliverable is part of the second key phase of the project. The stakeholders involved are the following partners:

- NBS System
- CNRS/LAAS
- UGA-LIG-ERODS

2 Objectives

- Provide tools capable of monitoring the status of Cloud and Edge infrastructures in real time.
- Quickly detect hardware or software failures, as well as security incidents (cyberattacks, operational anomalies).
- Improve the quality of service and resilience of distributed infrastructures.
- Rely on advanced techniques such as distributed tracing, time-series analysis, and artificial intelligence for incident identification and management.

3 Technical context

In Cloud-Edge environments, the complexity and distributed nature of resources make anomaly detection difficult. This deliverable aims to develop solutions capable of leveraging data from different layers (hardware, middleware, application) to improve anomaly detection and management, while optimizing energy efficiency and the distribution of monitoring techniques.

The illustration below (Figure 1) provides a high-level overview of the various components making up Work Package 2, segmented by functional layers. The components of this deliverable are located in the Monitoring/Security layer.

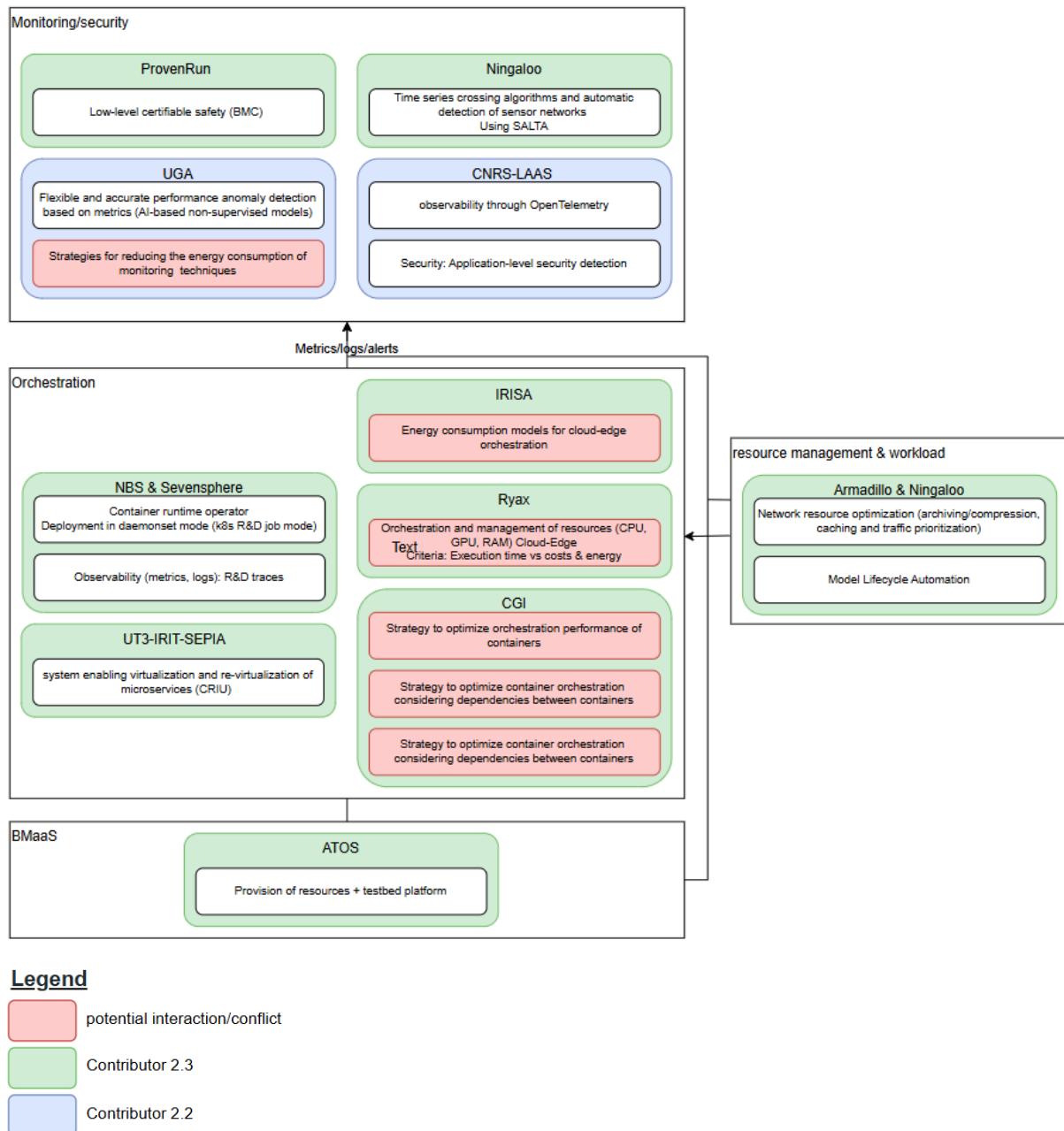


FIGURE 1 :HIGH-LEVEL DIAGRAM OF THE CLOUD-EDGE ORCHESTRATION SOLUTIONS

4 Restitution of contributors' contributions

4.1 CNRS-LAAS

4.1.1 Introduction

Microservices architecture has emerged as a solution to the limitations of traditional monolithic architectures, particularly in terms of scalability, flexibility, and resilience. Although no formal standard or single reference framework precisely defines this architecture, a consensus has gradually formed around its fundamental principles, contributing to its widespread adoption in industrial environments.

Today, microservices-oriented applications are generally deployed using the Kubernetes orchestrator [Kubernetes]. Since containers are constantly being created and destroyed, traditional monitoring—which is often limited to tracking basic resources like CPU or RAM—proves insufficient for detecting weak signals, isolating rare anomalies, or identifying the root cause of failures [Suchuan Xing et al. 2025].

To understand the exact path of a request, one solution is to use distributed tracing via standard tools such as OpenTelemetry [OpenTelemetry]. However, in Edge-type environments, which often have limited hardware resources, collecting and storing these massive volumes of traces quickly saturates the memory of local machines.

Two PhD students, Cheickna SYLLA and Léonard PAINTEAUX, are assigned to the project, the former is working on infrastructure-level monitoring solutions, and the latter on runtime security vulnerability detection. For the remainder of this report, their work will be presented in two separate sections.

The first contribution is therefore to design and deploy a telemetry pipeline capable of collecting traces and metrics from a microservices application, then offloading them to an optimized database. This approach offers several advantages, including reducing the load on runtime nodes, and can also serve as a starting point for creating datasets to train the anomaly detection algorithms that will be implemented.

To elaborate on this approach, this report will present the challenges associated with data processing in distributed environments, followed by the deployment of the otel-demo [OpenTelemetry Demo] test application with various test scenarios. It will then describe the offloading of traces to MinIO [MinIO] object storage, and finally the generation of controlled network anomalies via the Chaos Mesh [Chaos Mesh] tool, in order to demonstrate the value of tracing and highlight the limitations of traditional monitoring tools such as Prometheus [Prometheus].

The second contribution involves using the trace and metric retrieval pipeline to ensure application security. The main objective is to detect and handle anomalies and attacks as quickly as possible while keeping the impact on application performance to a minimum.

4.1.2 Challenges

- **Monitoring**

While traditional metric-based monitoring is useful for monitoring hardware status, it has a major flaw in distributed systems, it provides no context. If a transaction traverses ten microservices and slows down, a simple CPU alert on a node does not allow for the precise identification of the faulty component. The system then becomes vulnerable to gray outages, i.e., silent degradations where a service responds slowly without actually stopping.

Although distributed tracing allows for the explicit mapping of these requests' paths, it introduces a new technical challenge, its hardware cost. As demonstrated by a study evaluating the impact of telemetry on microservices [Anders, 2024], the process of configuring and exporting traces significantly degrades the performance of the monitored application. Tests in this study reveal drops in available throughput of up to 80% without prior sampling during data export. Furthermore, when initializing new containers, the load associated with the tracing tool generates latency spikes of up to 175%.

In architectures deployed at the network edge [Satyanarayanan, 2017], where computing power and storage are by definition very limited, this cost becomes critical. Asking these constrained nodes to manage, process, and store a continuous stream of observability data locally risks quickly saturating disks and bandwidth. The monitoring tool then risks becoming the very cause of instability. Faced with this, it is therefore imperative to decouple data collection from storage [OpenTelemetry Arch] in accordance with OpenTelemetry's recommendations, by exporting these streams asynchronously to an optimized external analytics environment.

However, this need to offload storage raises a security issue. Since telemetry streams are not merely performance metrics, they may contain sensitive data (identifiers, access tokens, query parameters) [Hannousse & Yahiouche, 2021]. The continuous export of these volumes from nodes to a centralized environment significantly expands the attack surface, exposing data in transit and creating a single point of failure that is particularly vulnerable to exfiltration [Shi et al., 2016]. One approach is to avoid sending raw data to a remote server, but instead move the analysis capability as close as possible to the source to extract insights locally [Lim et al., 2020].

- **Security**

Extensive research has been conducted on the potential security risks associated with the transition from monolithic architectures to microservice architectures. Beahan et al. [Beahan et al. 2025] analyzed 30 open-source projects based on microservices using three vulnerability scanning tools and identified a significant number of security flaws stemming primarily from third-party dependencies and container configurations. The most common vulnerabilities involved request and data handling, which could lead to denial-of-service attacks. In contrast, code-level vulnerabilities, although less frequent, have potentially higher severity and impact. Overall, the majority of identified vulnerabilities are of medium to high severity, underscoring the importance of secure data management, proper container configuration, and regular updates to dependencies. Minna et al. [Minna & Massacci, 2023] show that, despite widespread adoption of microservices, their runtime security remains understudied, despite the increased attack surface resulting from their distributed and poorly isolated nature. The study highlights

significant gaps regarding orchestration security, security policy verification, and the control of information flows between microservices. It also emphasizes that many existing solutions do not effectively test the security of deployed systems or vulnerabilities present in real-world environments. The authors conclude that runtime microservice security is still in its infancy and that future research should focus on reproducible testbeds and real-world security studies.

The vast majority of research on intrusion detection in cloud architectures relies on methods based on artificial intelligence. Hizal et al. [Hizal et al., 2021] used a model combining convolutional neural networks and GRU (Gated Recurrent Unit) recurrent layers to learn patterns from network traffic data. Using the NSL-KDD dataset, which contains network and protocol flows labeled as normal traffic or various attack categories (DoS – Denial of Service, Probe – reconnaissance/scan, R2L – Remote to Local, and U2R – User to Root), their system achieved high performance, with 99.86% accuracy for multi-class classification. The model is capable of effectively detecting and blocking unauthorized or malicious traffic before it reaches cloud servers, even in the context of imbalanced data. Chang et al. [Chang et al., 2019] addressed the problem of identifying and classifying microservices in application-aware data centers by proposing a non-intrusive fingerprinting approach based on runtime behavior. They combined lightweight system call tracing via eBPF with machine learning models incorporating Bayesian learning and LSTM (Long Short-Term Memory) autoencoders to accurately characterize microservices. The proposed method achieves up to 99% accuracy with minimal CPU overhead and enables the detection of previously unknown microservices with near-perfect accuracy. The study shows that the combination of Bayesian and LSTM models outperforms the use of each model in isolation, particularly when distinguishing between similar microservices and detecting outliers.

The results found in the literature, while excellent on the surface, should be viewed with caution. A primary limitation stems from the data used to train and test these models. Indeed, data generated in controlled environments are not always representative of reality. The models demonstrate very high success rates on this data, but there is no guarantee that these same models can ensure satisfactory detection during real-world operations. A second limitation of the results found in the literature is the lack of integration of the proposed solutions at runtime. Indeed, the overwhelming majority of studies consider only detection performance without taking into account the models' convergence time. However, in highly dynamic cloud environments, it is imperative to detect anomalies and attacks as quickly as possible. Major challenges then arise, such as model-induced latency, resource consumption, and the ability to scale in dynamic and highly distributed environments.

4.1.3 Work Carried out

- **Monitoring**

The overall architecture of the monitoring system, modeled in Figure 2, forms the basis of the experiments conducted on the Kubernetes cluster. This diagram illustrates the end-to-end telemetry chain, highlighting the integration of the various tools and the nature of the network flows circulating within it.

The test application deployed on our cluster is OpenTelemetry Astronomy Shop (otel-demo) [OTEL-DEMO]. It is a platform that simulates an e-commerce site composed of a dozen microservices written in Go, Python, Java, Node.js, and C++. All components natively embed the OpenTelemetry SDKs and APIs, ensuring the continuous and automatic emission of observability signals.

This infrastructure is structured around three main pillars, which will be discussed in the following subsections.

System stimulation and disruption: the generation of realistic user traffic via Locust is coupled with the injection of controlled failures directly onto the execution nodes using Chaos Mesh.

- Collection and processing: the interception of trace and metric signals by the application SDK is followed by their filtering and enrichment through the addition of Kubernetes metadata, operations handled by a local OpenTelemetry agent deployed as a DaemonSet.
- Export and analysis: asynchronous routing of telemetry data occurs to dedicated storage solutions—namely Jaeger, Prometheus, and MinIO—for analytical use via the Grafana interface.

This overview maps the exact path of the information, from the application to its persistence, and provides an optimal understanding of the observability mechanisms detailed below.

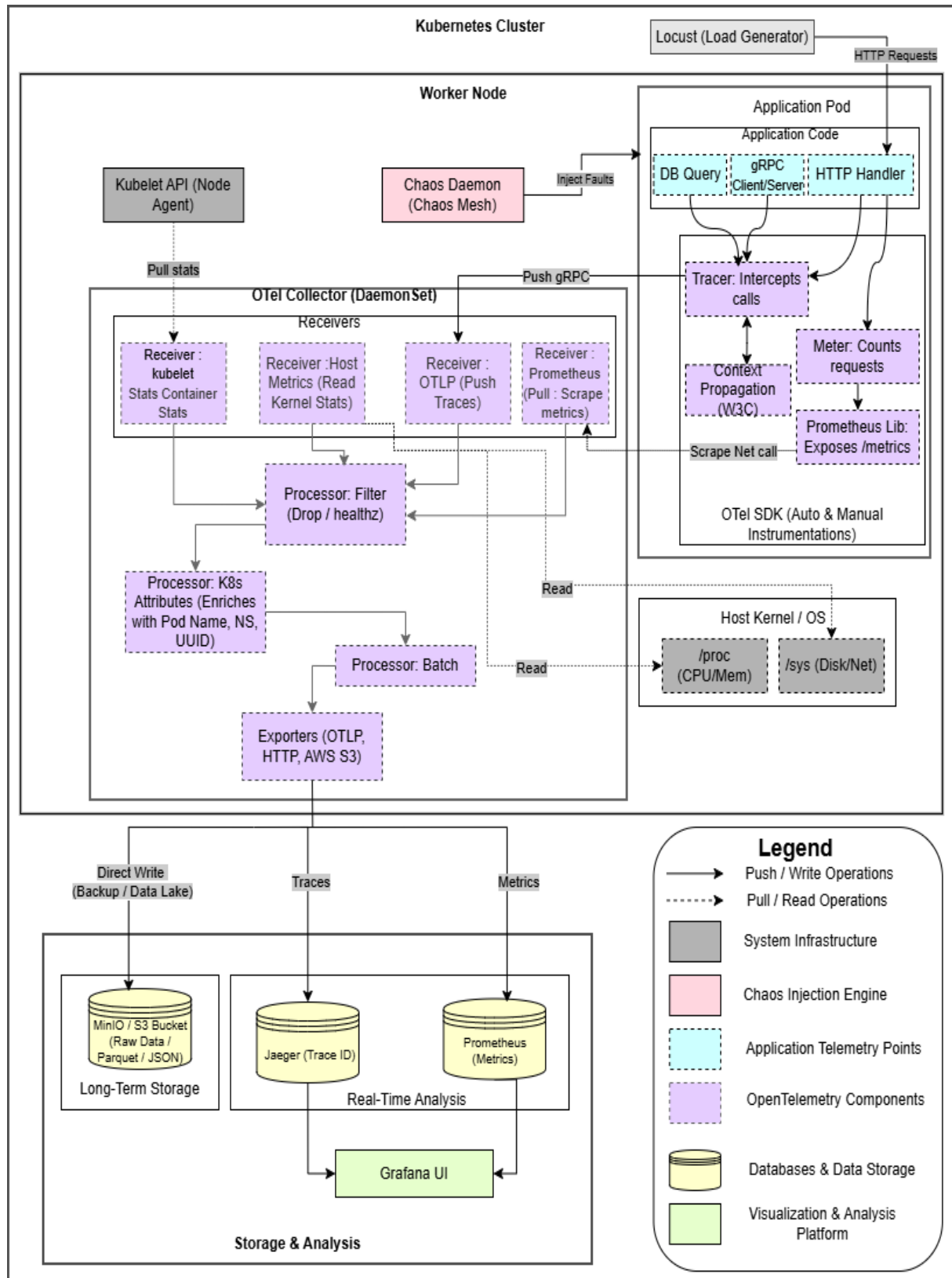


FIGURE 2 : OVERALL ARCHITECTURE OF THE MONITORING SYSTEM IMPLEMENTED

- **OpenTelemetry Standard and Trace Representation**

The instrumentation and standardization of signal acquisition within the scope of this work are based on the OpenTelemetry framework. This standard defines a trace as the representation of a request's execution path throughout the entire system [OpenTelemetry API]. A trace is modeled as a directed graph whose nodes are called Spans. Each Span represents a logical unit of execution, such as an HTTP request or a database transaction. It encapsulates metadata in the form of attributes (key-value pairs), allowing parameters such as the service name or the associated HTTP return code to be recorded.

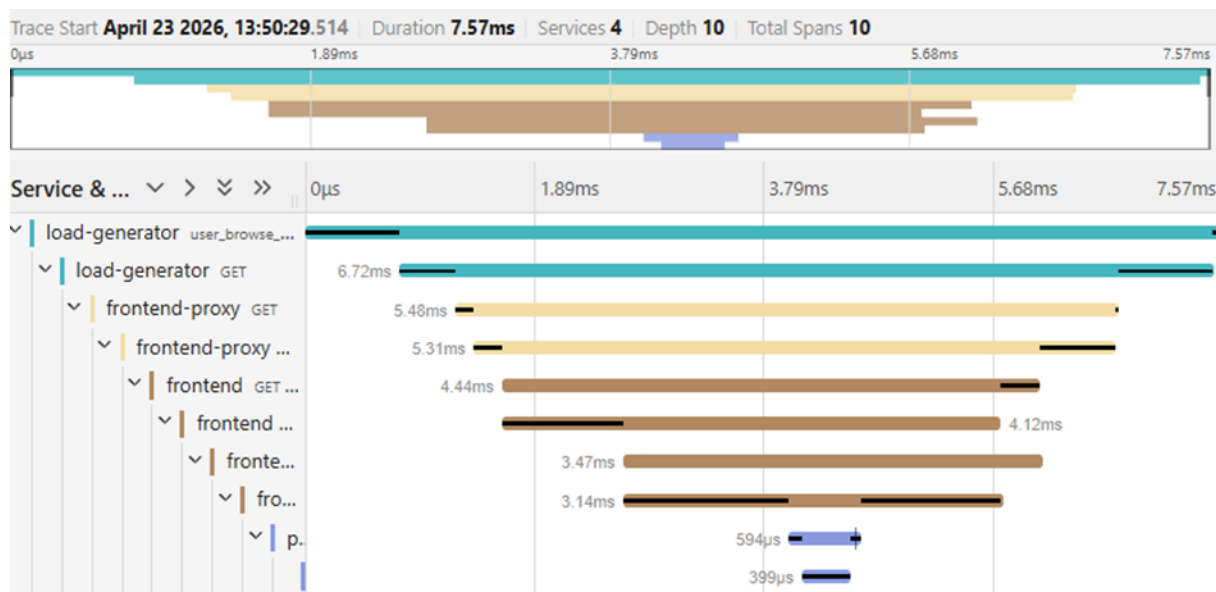


FIGURE 3 : GANTT CHART REPRESENTATION OF A DISTRIBUTED TRACE (JAEGER INTERFACE)

Figure 3 presents a Gantt chart of a trace captured via the Jaeger analysis interface [Jaeger]. This temporal visualization shows how a request initiated here by the load-generator component is subdivided. It shows the end-to-end propagation of the context; the transaction traverses 4 distinct services and is divided into 10 successive spans, with a total execution time of 7.57 ms. It is this hierarchical data structure that we aim to collect at scale.

- **Overview of the Experimental Platform**

The tests were conducted on the SLICES-RI research infrastructure, which is a flexible platform designed to support large-scale experimental research focused on network protocols, radio technologies, services, data collection, parallel and distributed computing, and in particular cloud and edge computing architectures and services [SLICES-RI].

SLICES provides command-line tools (CLI) based on Python virtual environments. These tools allow interaction with the infrastructure's physical resources. We used them to initially deploy five virtual machines using the following command:

```
(slices-venv) chsylla@LAPTOP-SYLLA:~$ slices bi --site-id be-gent1-bi-vm1 create vms --
experiment testkubes --image "Ubuntu 24.04.3" --flavor medium --duration 1d --
count=5 --public-ipv4 --ssh-key-file /mnt/c/Users/chsylla/.ssh/id_rsa.pub
```

(slices-venv) chsylla@LAPTOP-SYLLA:~\$ slices bi flavor list
Slices BI VM Flavors

Infra	Name	Desc	(v)CPUs	RAM (GiB)	Root Disk (GB)
be-gent1-bi-vm1	tiny	Minimal VM	1	1	4
be-gent1-bi-vm1	m1.small	Small General Purpose VM	1	2	10
be-gent1-bi-vm1	small	Small General Purpose VM	2	4	20
be-gent1-bi-vm1	medium	Medium General Purpose VM	4	8	50
be-gent1-bi-vm1	bdt-vm	VM for BDT course	4	8	75
be-gent1-bi-vm1	large	Large General Purpose VM	4	16	100
be-gent1-bi-vm1	xlarge	Extra Large General Purpose VM	8	32	200

Slices BI Baremetal Flavors

Infra	Name	Desc	CPUs	RAM (GiB)	Root Disk (GB)	Interfaces	GPU
be-gent1-bi-baremetal1	pc	Any baremetal pc	0x0x0	0	0	0	-
be-gent1-bi-baremetal1	pcgen05-gpu5090	Baremetal generation 5 with GPU GTX5090	1x8x1	32	250	0	GTX5090
be-gent1-bi-baremetal1	pcgen07	Baremetal generation 7	1x6x1	64	512	0	-
be-gent1-bi-baremetal1	pcgen07-gpu1080	Baremetal generation 7 with GPU GTX1080	1x6x1	64	512	0	GTX1080
be-gent1-bi-baremetal1	pcgen07-gpu980	Baremetal generation 7 with GPU GTX980	1x6x1	64	512	0	GTX980
be-gent1-bi-baremetal1	pcgen08	Baremetal generation 8 with GPU GTX5090	1x8x1	500.75	250	0	GTX5090

FIGURE 4 : LIST OF RESOURCES PROVIDED BY SLICES-RI

With the following configurations per VM (virtual machine):

Operating system: Linux Ubuntu 24.04.3

Number of vCPUs: 4, corresponding to the medium flavor (Figure 4)

RAM: 8 GB

Disk space: 50 GB.

To view all resources deployed on SLICES, use the following command:

```
(slices-venv) chsylla@LAPTOP-SYLLA:~$ slices bi list-resources --experiment testkube
```

The result of the command is shown in Figure 5.

(slices-venv) chsylla@LAPTOP-SYLLA:~\$ slices bi list-resources --experiment testkube
Resources in experiment exp_expauth.ilabt.imec.be_01kq8n2vndfmhrc1er1q2nca88

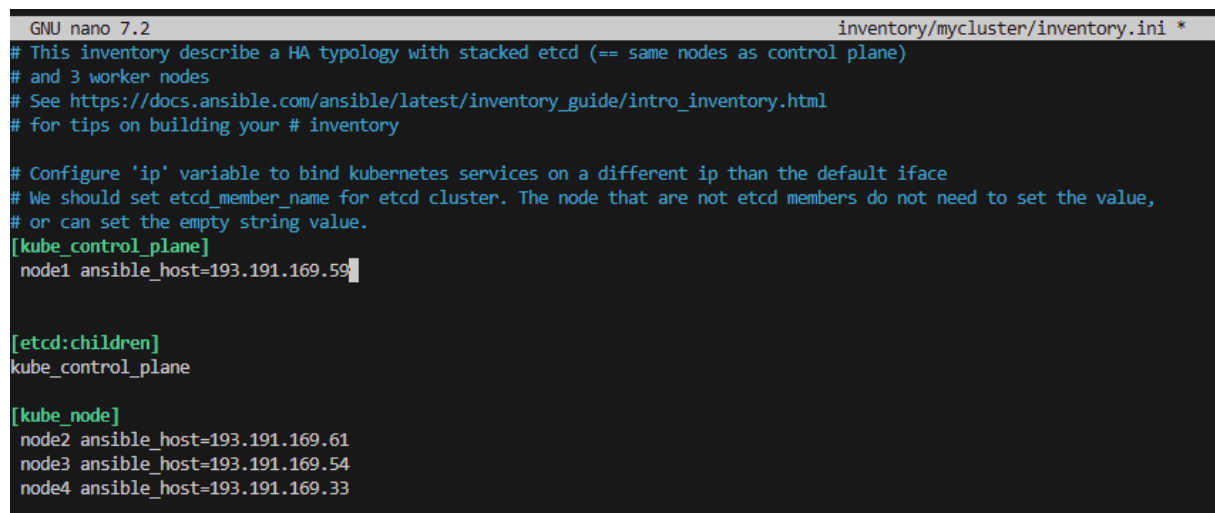
ID	Name	Status	Created At	Expires At	Public IPv4	Private IPv4
r_be-gent1-bi-vm1_01kq8n385ne219nebgrryx4q0c	vms0	up	2026-04-28 01:41 CEST	2026-05-05 01:41 CEST	193.191.169.26	
r_be-gent1-bi-vm1_01kq8n385ne82axgcjssjhf6f6	vms1	up	2026-04-28 01:41 CEST	2026-05-05 01:41 CEST	193.191.169.59	
r_be-gent1-bi-vm1_01kq8n385neng82j19yepy7wx8	vms2	up	2026-04-28 01:41 CEST	2026-05-05 01:41 CEST	193.191.169.61	
r_be-gent1-bi-vm1_01kq8n385nf5k91pss62a9byx0	vms3	up	2026-04-28 01:41 CEST	2026-05-05 01:41 CEST	193.191.169.54	
r_be-gent1-bi-vm1_01kq8n385nf4ksr5fafjbayp9g	vms4	up	2026-04-28 01:41 CEST	2026-05-05 01:41 CEST	193.191.169.33	

FIGURE 5 : STATUS OF CURRENTLY RUNNING VM 1

One of these VMs will serve as a control machine to instantiate a Kubernetes cluster on the remaining virtual machines.

This control VM uses the open-source tool Kubespray [Kubespray], which leverages Ansible to automate the deployment, configuration, and management of production Kubernetes clusters on any type of infrastructure.

Available on GitHub, Kubespray orchestrates the deployment by running an Ansible playbook (cluster.yml) via SSH to install the cluster components. The project relies on preconfigured global variables (inventory/mycluster/group_vars/) such as the Kubernetes version, the runtime type, and the range of virtual IP addresses reserved for Kubernetes services for internal routing; we only had to modify the inventory file (inventory/mycluster/hosts.yaml) shown in Figure 6 to declare the IP addresses and roles of our target machines [Kubespray].



```
GNU nano 7.2 inventory/mycluster/inventory.ini *
# This inventory describe a HA typology with stacked etcd (== same nodes as control plane)
# and 3 worker nodes
# See https://docs.ansible.com/ansible/latest/inventory_guide/intro_inventory.html
# for tips on building your # inventory

# Configure 'ip' variable to bind kubernetes services on a different ip than the default iface
# We should set etcd_member_name for etcd cluster. The node that are not etcd members do not need to set the value,
# or can set the empty string value.
[kube_control_plane]
node1 ansible_host=193.191.169.59

[etcd:children]
kube_control_plane

[kube_node]
node2 ansible_host=193.191.169.61
node3 ansible_host=193.191.169.54
node4 ansible_host=193.191.169.33
```

FIGURE 6 : KUBERNETES NODE CONFIGURATION IN THE HOST.YAML INVENTORY FILE

Once this file is configured, the following command is run:

```
ansible-playbook -i inventory/mycluster/inventory.ini -b cluster.yml
```

This will allow Ansible to retrieve the inventory file to identify the role of each machine (the master nodes and workers). Using the `-b` (become) argument, the script elevates its privileges to obtain root administrator rights on all targets. Ansible then connects via SSH to each server and reads the main cluster.yml file, which contains all the directives that are installed sequentially, namely system preparation, network routing configuration, installation of the execution engine (containerd), and initialization of Kubernetes components (kubeadm, kubelet, etc.) whose preconfigured configuration file is shown in Figure 7:

```
GNU nano 7.2 cluster.yml
--
- name: Common tasks for every playbooks
  import_playbook: boilerplate.yml

- name: Gather facts
  import_playbook: internal_facts.yml

- name: Prepare for etcd install
  hosts: k8s_cluster:etcd
  gather_facts: false
  any_errors_fatal: "{{ any_errors_fatal | default(true) }}"
  environment: "{{ proxy_disable_env }}"
  roles:
    - { role: kubenspray_defaults }
    - { role: kubernetess/preinstall, tags: preinstall }
    - { role: "container-engine", tags: "container-engine", when: deploy_container_engine }
    - { role: download, tags: download, when: "not skip_downloads" }

- name: Install etcd
  vars:
    etcd_cluster_setup: true
    etcd_events_cluster_setup: "{{ etcd_events_cluster_enabled }}"
  import_playbook: install_etcd.yml

- name: Install Kubernetes nodes
  hosts: k8s_cluster
  gather_facts: false
  any_errors_fatal: "{{ any_errors_fatal | default(true) }}"
  environment: "{{ proxy_disable_env }}"
  roles:
    - { role: kubenspray_defaults }
    - { role: kubernetess/node, tags: node }

- name: Install the control plane
  hosts: kube_control_plane
  gather_facts: false
  any_errors_fatal: "{{ any_errors_fatal | default(true) }}"
  environment: "{{ proxy_disable_env }}"
  roles:
    - { role: kubenspray_defaults }
    - { role: kubernetess/control-plane, tags: control-plane }
    - { role: kubernetess/client, tags: client }
    - { role: kubernetess/apps/cluster_roles, tags: cluster-roles }
```

FIGURE 7 : CLUSTER.YML CONFIGURATION FILE PROVIDED BY KUBESPRAY [KUBESPRAY]

More specifically, this Ansible manifest segments this deployment into blocks of instructions applied specifically according to the role assigned to each target machine. After initializing the environment and the runtime engine via the container-engine role, the script proceeds to install etcd, the distributed key-value database essential for centrally storing and synchronizing the system's global configuration. It then deploys the necessary components on the compute nodes (associated with the hosts: k8s_cluster group) so that they can host future microservices. The architecture is finally completed by initializing the control plane on the master node (hosts: kube_control_plane), which deploys the central management components and enforces security and network access policies by executing the administration roles (cluster-roles).

After installation and according to our configuration defined in Figures 6 and 7, the Kubernetes cluster is created with the following information, as shown in Figure 8:

```
ubuntu@node1:~$ sudo kubectl get nodes -o wide
```

NAME	STATUS	ROLES	AGE	VERSION	INTERNAL-IP	EXTERNAL-IP	OS-IMAGE	KERNEL-VERSION	CONTAINER-RUNTIME
node1	Ready	control-plane	11h	v1.35.4	193.191.169.59	<none>	Ubuntu 24.04.3 LTS	6.8.0-85-generic	containerd://2.2.3
node2	Ready	<none>	11h	v1.35.4	193.191.169.61	<none>	Ubuntu 24.04.3 LTS	6.8.0-85-generic	containerd://2.2.3
node3	Ready	<none>	11h	v1.35.4	193.191.169.54	<none>	Ubuntu 24.04.3 LTS	6.8.0-85-generic	containerd://2.2.3
node4	Ready	<none>	11h	v1.35.4	193.191.169.33	<none>	Ubuntu 24.04.3 LTS	6.8.0-85-generic	containerd://2.2.3

FIGURE 8 : KUBERNETES CLUSTER

This information confirms the overall status of the cluster. It shows that the four VMs have been successfully instantiated with a Ready status. The node1 machine has the control-plane role to orchestrate the system, while the node2, node3, and node4 machines operate as compute nodes. The display also details the software stack across the entire network; each machine runs Kubernetes version v1.35.4 on Ubuntu 24.04.3 LTS with the containerd runtime.

- **Experimentation with Otel-demo**

The test application deployed on our cluster is OpenTelemetry Astronomy Shop (otel-demo) [OTEL-DEMO]. It is a platform that simulates an e-commerce site composed of a dozen microservices written in Go, Python, Java, Node.js, and C++. All components natively embed the OpenTelemetry SDKs and APIs, ensuring the continuous and automatic emission of observability signals.

To illustrate this mechanism, let us examine the critical purchasing flow. When a user interacts with the Frontend, it communicates with the Checkout Service. This service then orchestrates the end-to-end transaction by simultaneously interrogating the various backends (Payment, Shipping, Email, as well as Cart and Currency), primarily using the gRPC protocol, as shown in Figure 9.

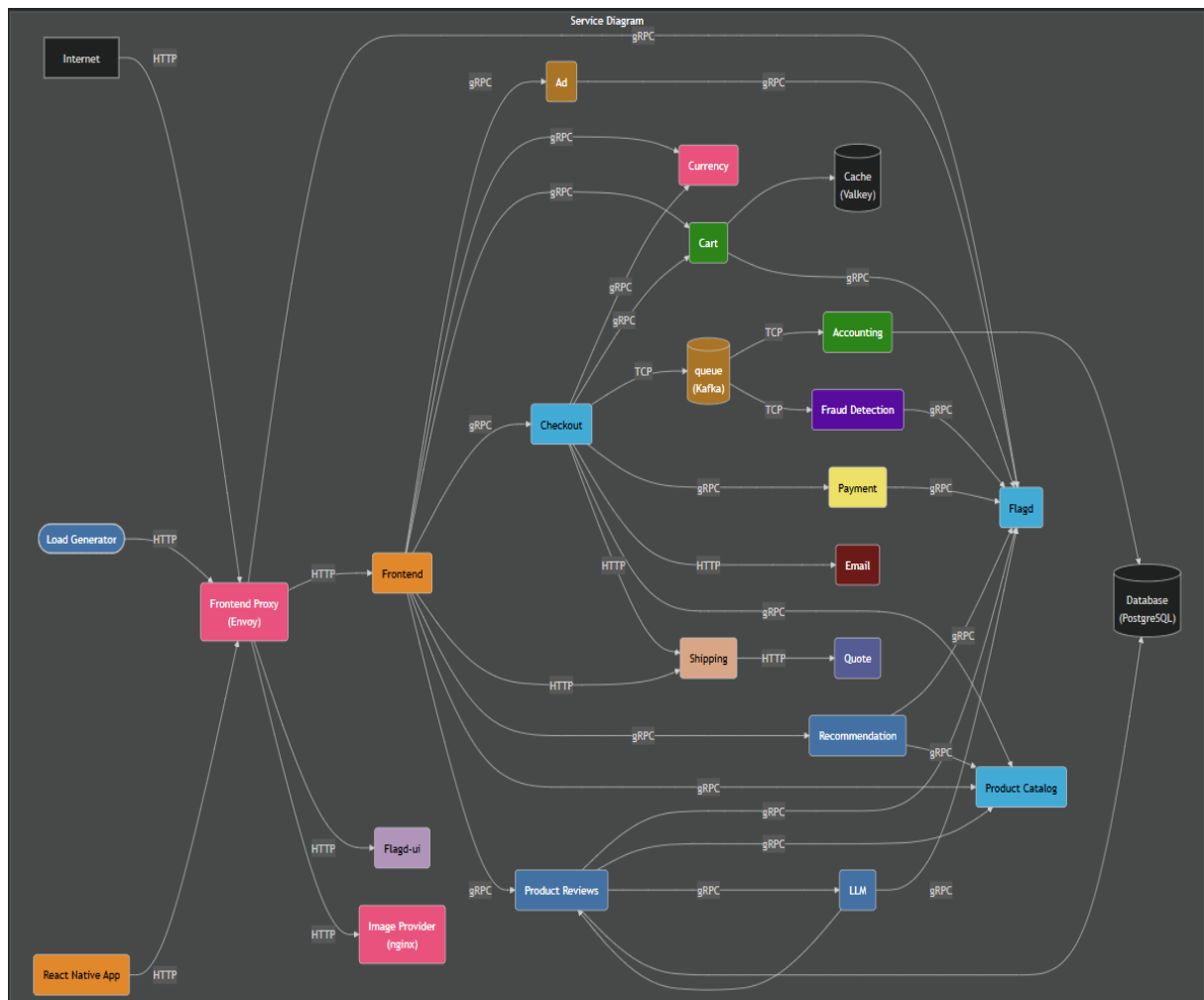


FIGURE 9 : OPERATIONAL ARCHITECTURE OF THE OTEL-DEMO APPLICATION [OTEL-DEMO]

The main components of the application are:

- Load Generation with Locust

To validate our platform, we need to simulate realistic traffic. This is accomplished by the Load Generator component, which relies on Locust [Locust], an open-source load testing framework written in Python.

It executes a file called `locustfile.py` [Locustfile] in which user scenarios are coded. As shown in the code snippet in Figure 10, the behavior is probabilistic and defined by weights directly visible within the `@task` decorators. Thus, the task of viewing a product (`browse_product`) has a weight of 10, making it much more frequent than viewing the cart (`view_cart`), whose weight is set to 3.

```
class WebsiteUser(HttpUser):
    wait_time = between(1, 10)

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.tracer = trace.get_tracer(__name__)

    @task(1)
    def index(self): ...

    @task(10)
    def browse_product(self):
        product = random.choice(products)
        with self.tracer.start_as_current_span("user_browse_product", context=Context(), attributes={"product.id": product}):
            logging.info(f"User browsing product: {product}")
            self.client.get("/api/products/" + product)

    @task(3)
    def get_recommendations(self): ...

    @task(2)
    def get_product_reviews(self): ...

    @task(1)
    def ask_product_ai_assistant(self): ...

    @task(3)
    def get_ads(self): ...

    @task(3)
    def view_cart(self):
        with self.tracer.start_as_current_span("user_view_cart", context=Context()):
            logging.info("User viewing cart")
            self.client.get("/api/cart")
```

FIGURE 10 : EXCERPT FROM THE LOAD GENERATION SCRIPT LOCUSTFILE.PY PROVIDED BY OTEL-DEMO [LOCUSTFILE]

Locust uses a system of very lightweight asynchronous tasks via the `gevent` library to simulate hundreds of concurrent users from a single process. Figure 10 also shows that these virtual users, which inherit from the `HttpUser` class, generate actual HTTP network requests such as `self.client.get("/api/products/...")` that target the endpoints of the Frontend component in our cluster. We can also see the use of the OpenTelemetry API (`self.tracer.start_as_current_span`) to manually create a trace named `user_browse_product` that encompasses the HTTP request, and to inject specific attributes into it—such as the ID of the product being viewed—before sending it to the Collector.

This allows us to overload the application and observe how the microservices respond to the load.

- OpenTelemetry Collector

The central component is the OpenTelemetry Collector [OpenTelemetry Collector]. It acts as a three-phase processing pipeline, the implementation of which is demonstrated by its Kubernetes YAML configuration file `otel-collector-agent` (Figure 11)

- Receivers: The Collector instantiates an OTLP (OpenTelemetry Protocol) receiver that listens on specific ports (4317 for gRPC, 4318 for HTTP) to accept raw data sent by the microservices.
- Processors: This is where the data is processed according to the conditions defined in the configuration files.

- Filtering: The Collector can use a filtering processor to eliminate unnecessary traces at the source, such as /healthz health check requests generated by Kubernetes, to prevent network and storage saturation.
- Metadata: The architecture relies on the k8sattributes processor [K8s Attributes Processor]. This component queries the Kubernetes API in real time via a role-based access control (RBAC) mechanism [K8s RBAC] to attach labels to raw signals. Thus, an isolated metric is automatically enriched with the exact name of the Pod (pod.name), the physical node hosting it (k8s.node.name), and its namespace.
- Exporters: Once the data is formatted and enriched, the Collector pipeline translates it and routes it to the visualization backends.

```

apiVersion: v1
kind: ConfigMap
metadata:
  name: otel-collector-agent
  namespace: otel-demo
data:
  relay: |
    receivers:
      otlp:
        protocols:
          grpc:
            endpoint: ${env:MY_POD_IP}:4317
          http:
            endpoint: ${env:MY_POD_IP}:4318

    processors:
      k8sattributes:
        extract:
          metadata:
            - k8s.namespace.name
            - k8s.pod.name
            - k8s.node.name
            ....
        filter:
          node_from_env_var: K8S_NODE_NAME

    exporters:
      otlp/jaeger:
        endpoint: jaeger:4317
      otlphttp/prometheus:
        endpoint: http://prometheus:9090/api/v1/otlp
      # ...

  service:
    pipelines:
      traces:
        receivers: [otlp, jaeger, zipkin]
        processors: [k8sattributes, memory_limiter, resourcedetection, resource, transform, batch]
        exporters: [otlp/jaeger, debug, spanmetrics]
      metrics:
        receivers: [otlp, kafkametrics, spanmetrics, ...]
        processors: [k8sattributes, memory_limiter, resourcedetection, ...]
        exporters: [otlphttp/prometheus, debug]

```

FIGURE 11 : SUMMARY EXCERPT FROM THE OTEL-COLLECTOR-AGENT CONFIGMAP WITH THE OBSERVABILITY PIPELINE IMPLEMENTATION (RECEIVERS, PROCESSORS, EXPORTERS) [PROMETHEUS].

- Data Storage, Visualization, and Analysis

In the otel-demo application, once the data has been processed by the OpenTelemetry Collector, it is redirected to open-source analytics solutions. This export mechanism is defined in the Collector's configuration file (Figure 11), within the exporters block. The separation of storage backends is effective because traces and metrics have different data structures.

Prometheus for metrics [Prometheus]: This is a monitoring and alerting system that functions as a time-series database. The Collector continuously pushes data to it via the `otlphttp/prometheus` block (Figure 11), which targets its API. In our cluster, it aggregates all quantitative data (CPU/RAM usage, HTTP error rates). Its data model allows it to query these metrics via its native query language, PromQL.

Jaeger for distributed traces [Jaeger]: This is a platform dedicated to tracing transactions in distributed systems. The Collector communicates with it via the gRPC protocol defined in `otlp/jaeger` (Figure 11). Jaeger receives the spans and visually reconstructs the complete execution tree of a request.

Grafana for global monitoring [Grafana Labs]: Published by Grafana Labs, Grafana is an analytics and visualization platform. In terms of software architecture, Grafana does not directly receive data pushed by the Collector; it reads it from the source. To automate this connection during Helm deployment [Grafana Labs], the infrastructure relies on a provisioning mechanism. Specifically, Kubernetes injects a configuration file (ConfigMap) directly into the Grafana container (Figure 12). This file statically declares Prometheus and Jaeger as data sources, allowing the interface to query their respective APIs in real time over the internal network to build dashboards.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: grafana-datasources
  namespace: otel-demo
data:
  default.yaml: |
    apiVersion: 1
    datasources:
      - name: Prometheus
        type: prometheus
        url: http://prometheus:9090
        isDefault: true

  jaeger.yaml: |
    apiVersion: 1
    datasources:
      - name: Jaeger
        type: jaeger
        url: http://jaeger:16686/jaeger/ui
        isDefault: false
```

FIGURE 12 : EXCERPT FROM THE GRAFANA-DATASOURCES CONFIGMAP [GRAFANA DATASOURCES]

- **Deploying otel-demo on the cluster**

To orchestrate the deployment of all microservices and the observability stack on our cluster, we used the Helm tool [Grafana Labs], which acts as the standard package manager for Kubernetes. It automates the deployment, configuration, and management of complex applications by grouping Kubernetes manifests into preconfigured templates called Charts.

The application was launched from the master node using the following command. This command instantiates the official OpenTelemetry chart in a dedicated namespace named otel-demo (Figure 13)

```
sudo helm install my-otel-demo open-telemetry/opentelemetry-demo -n otel-demo
```

```
ubuntu@node1:~$ sudo helm install my-otel-demo open-telemetry/opentelemetry-demo -n otel-demo
NAME: my-otel-demo
LAST DEPLOYED: Thu Apr 30 16:00:21 2026
NAMESPACE: otel-demo
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
=====
OTEL DEMO

- All services are available via the Frontend proxy: http://localhost:8080
  by running these commands:
    kubectl --namespace otel-demo port-forward svc/frontend-proxy 8080:8080

The following services are available at these paths after the frontend-proxy service is exposed with port forwarding:
Webstore          http://localhost:8080/
Jaeger UI         http://localhost:8080/jaeger/ui/
Grafana           http://localhost:8080/grafana/
Load Generator UI http://localhost:8080/loadgen/
Feature Flags UI  http://localhost:8080/feature/
```

FIGURE 13 : RESULT OF EXECUTING THE HELM COMMAND, CONFIRMING THAT THE CLUSTER HAS PROCESSED THE DEPLOYMENT OF THE OPENTELEMETRY-DEMO CHART.

Once the deployment order has been sent to the Kubernetes API, the kubectl command-line utility can be used to verify that the resources have been successfully instantiated. Figure 14 confirms that each microservice, database, and telemetry tool has started correctly. The NAME column identifies the deployed component. The READY column specifies the ratio of operational containers within a single Pod, illustrated by complete values such as 4/4 for the Grafana tool or 2/2 for the flagd service. The STATUS column confirms that the entire infrastructure has reached the Running execution state. Finally, the RESTARTS column tracks successive restarts to assess the system's immediate stability, highlighting here only a double restart on the product-catalog service during its initialization.

```
ubuntu@node1:~$ sudo kubectl get pods -n otel-demo
```

NAME	READY	STATUS	RESTARTS	AGE
accounting-5ccf5cf965-xrjkg	1/1	Running	0	9m13s
ad-5547bd5bd9-64pdf	1/1	Running	0	9m15s
cart-74f5bf48d9-5xbc7	1/1	Running	0	9m12s
checkout-546dd7dbd9-jvrvv	1/1	Running	0	9m12s
currency-5c757d5f4-g54jh	1/1	Running	0	9m11s
email-78fcc8d79-sh549	1/1	Running	0	9m15s
flagd-5ff58bc756-wrfh9	2/2	Running	0	9m12s
fraud-detection-74c96487b7-4qmw9	1/1	Running	0	9m12s
frontend-6bd6cb5df4-t44h9	1/1	Running	0	9m13s
frontend-proxy-697fbd5699-b7p2g	1/1	Running	0	9m15s
grafana-78d45767bc-cv62n	4/4	Running	0	9m14s
image-provider-76655d4b8d-mf8pz	1/1	Running	0	9m13s
jaeger-6799894d46-2c2vt	1/1	Running	0	9m14s
kafka-84d959f89d-qrxjx	1/1	Running	0	9m15s
llm-fb88677f7-66vwm	1/1	Running	0	9m14s
load-generator-d8b6686fb-7cfnx	1/1	Running	0	9m14s
opensearch-0	1/1	Running	0	9m15s
otel-collector-agent-bl76j	1/1	Running	0	9m15s
otel-collector-agent-jfbds	1/1	Running	0	9m15s
otel-collector-agent-x7cfw	1/1	Running	0	9m15s
payment-5c5df75598-qbgmt	1/1	Running	0	9m15s
postgresql-66fff77f5d-md4gn	1/1	Running	0	9m15s
product-catalog-d6db96fb7-wjnm1	1/1	Running	2 (9m14s ago)	9m15s
product-reviews-5996cd4c78-xpw88	1/1	Running	0	9m14s
prometheus-685b54b847-n6msj	1/1	Running	0	9m13s
quote-6cdc7d568f-df5fl	1/1	Running	0	9m14s
recommendation-78c689f8cf-c7h29	1/1	Running	0	9m15s
shipping-78f88699cd-xrtnw	1/1	Running	0	9m15s
valkey-cart-779654c7f4-t8rv4	1/1	Running	0	9m15s

FIGURE 14 : LIST AND STATUS OF PODS IN THE OTEL-DEMO NAMESPACE

To delve deeper into the analysis, we can observe the physical distribution of these Pods across our various virtual machines (Figure 15). This distribution is the result of the work performed by the kube-scheduler Suchuan Xing running on the master node to orchestrate the infrastructure. This native component of the Kubernetes control plane analyzes the available resources (CPU, RAM) on each worker node and dynamically assigns microservices to balance the load and ensure high availability of the application.

```
ubuntu@node1:~$ sudo kubectl get pods -o wide -n otel-demo
```

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE	NOMINATED	NODE	READINESS	GATES
accounting-5ccf5cf965-xrjpk	1/1	Running	0	12m	10.233.74.120	node4	<none>		<none>	
ad-5547bd5bd9-64pdf	1/1	Running	0	12m	10.233.71.2	node3	<none>		<none>	
cart-74f5bf48d9-5xbc7	1/1	Running	0	12m	10.233.74.121	node4	<none>		<none>	
checkout-546dd7dbd9-jvvrw	1/1	Running	0	12m	10.233.71.3	node3	<none>		<none>	
currency-5c757d5f4-g54jh	1/1	Running	0	12m	10.233.75.6	node2	<none>		<none>	
email-78fcc8d79-sh549	1/1	Running	0	12m	10.233.75.57	node2	<none>		<none>	
flagd-5ff58bc756-wrfh9	2/2	Running	0	12m	10.233.74.122	node4	<none>		<none>	
fraud-detection-74c96487b7-4qnw9	1/1	Running	0	12m	10.233.75.3	node2	<none>		<none>	
frontend-6bd6cb5df4-t44h9	1/1	Running	0	12m	10.233.75.63	node2	<none>		<none>	
frontend-proxy-697fbd5699-b7p2g	1/1	Running	0	12m	10.233.74.115	node4	<none>		<none>	
grafana-78d45767bc-cv62n	4/4	Running	0	12m	10.233.74.119	node4	<none>		<none>	
image-provider-76655d4b8d-mf8pz	1/1	Running	0	12m	10.233.71.9	node3	<none>		<none>	
jaeger-6799894d46-2c2vt	1/1	Running	0	12m	10.233.71.11	node3	<none>		<none>	
kafka-84d959f89d-grxjx	1/1	Running	0	12m	10.233.74.112	node4	<none>		<none>	
llm-fb88677f7-66vwm	1/1	Running	0	12m	10.233.74.116	node4	<none>		<none>	
load-generator-d8b6686fb-7cfnx	1/1	Running	0	12m	10.233.75.62	node2	<none>		<none>	
opensearch-0	1/1	Running	0	12m	10.233.75.61	node2	<none>		<none>	
otel-collector-agent-bl76j	1/1	Running	0	12m	10.233.74.113	node4	<none>		<none>	
otel-collector-agent-jfbsd	1/1	Running	0	12m	10.233.71.8	node3	<none>		<none>	
otel-collector-agent-x7cfw	1/1	Running	0	12m	10.233.75.59	node2	<none>		<none>	
payment-5c5df75598-qbgmt	1/1	Running	0	12m	10.233.75.58	node2	<none>		<none>	
postgresql-66fff77f5d-md4gn	1/1	Running	0	12m	10.233.71.13	node3	<none>		<none>	
product-catalog-d6db96fb7-wjnm1	1/1	Running	2 (12m ago)	12m	10.233.74.114	node4	<none>		<none>	
product-reviews-5996cd4c78-xpw88	1/1	Running	0	12m	10.233.74.118	node4	<none>		<none>	
prometheus-685b54b847-n6msj	1/1	Running	0	12m	10.233.71.5	node3	<none>		<none>	
quote-6cdc7d568f-df5f1	1/1	Running	0	12m	10.233.74.117	node4	<none>		<none>	
recommendation-78c689f8cf-c7h29	1/1	Running	0	12m	10.233.71.6	node3	<none>		<none>	
shipping-78f88699cd-xrtmw	1/1	Running	0	12m	10.233.75.60	node2	<none>		<none>	
valkey-cart-779654c7f4-t8rv4	1/1	Running	0	12m	10.233.71.12	node3	<none>		<none>	

FIGURE 15 : DISTRIBUTION OF MICROSERVICES ACROSS THE CLUSTER'S VARIOUS NODES, ILLUSTRATING THE LOAD-BALANCING MECHANISM IMPLEMENTED BY THE KUBE-SCHEDULER.

For security reasons, microservices deployed in a Kubernetes cluster do not directly expose their ports to the external public network. They use an internal mechanism called ClusterIP, which assigns them private IP addresses confined within the cluster. Thus, to access our platform's web interface locally, a two-step traffic redirection chain must be executed:

From the virtual machine hosting the master node, the following command is executed:

```
sudo kubectl --namespace otel-demo port-forward svc/frontend-proxy 8080:8080
```

This command specifically targets the component named frontend-proxy (Figure 15), which acts as an Envoy-based reverse proxy [Envoy Proxy], responsible for receiving user requests and distributing them to the appropriate microservices.

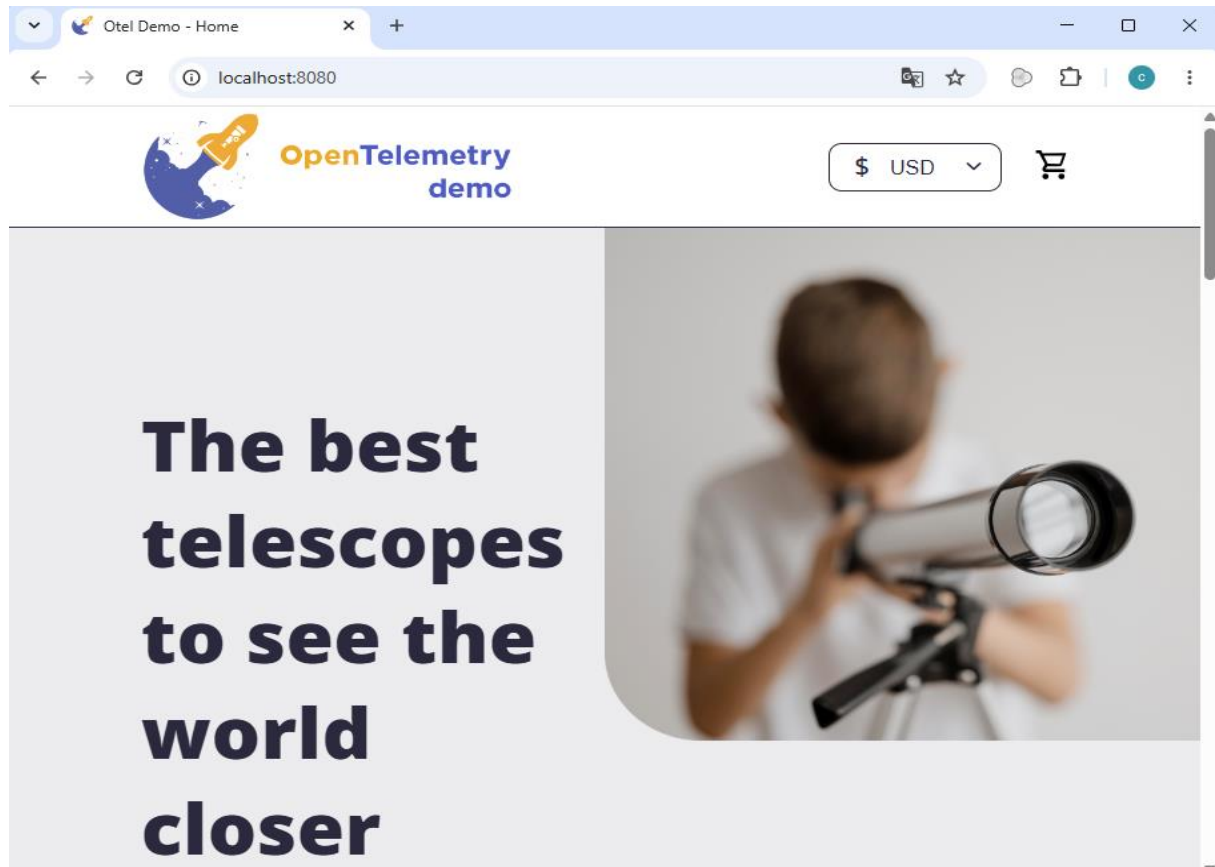
Port 8080 is the default listening port defined in the Helm package's [OTEL Helm Chart] application configuration files for accessing the store's web interface. The port-forward command therefore creates a temporary bridge linking this internal port 8080 to the master node's local port 8080.

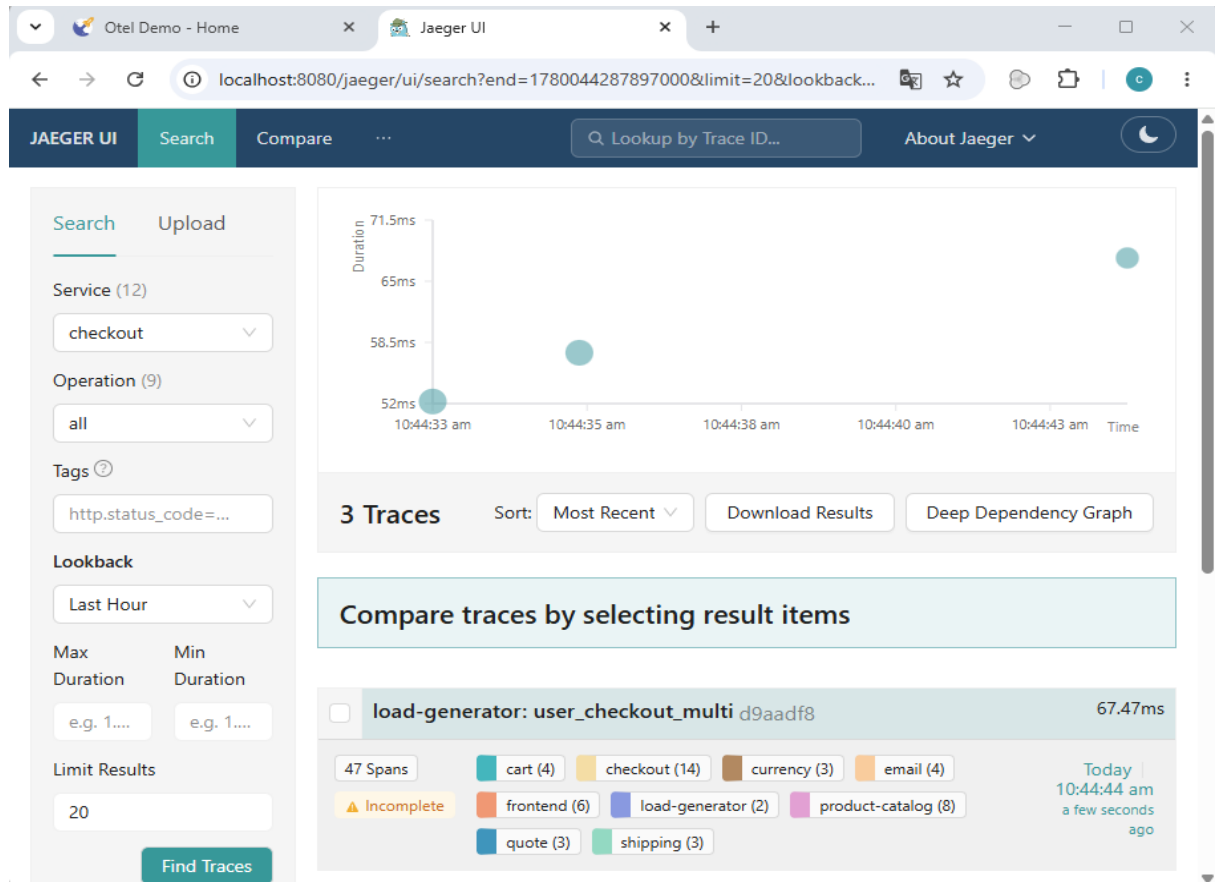
From the local PC terminal, we establish a network tunnel via SSH with:

```
ssh -N -f -L 8080:localhost:8080 -i "ssh_key" ubuntu@master_IP
```

This command captures all web traffic directed to its own port 8080, encapsulates it within the SSH tunnel, and delivers it to port 8080 on the remote master node.

Therefore, once we type the address <http://localhost:8080> (Figure 16) into our local web browser, the request travels through the encrypted SSH tunnel, arrives at the master machine, and is then forwarded by Kubernetes directly to the application's frontend-proxy service.





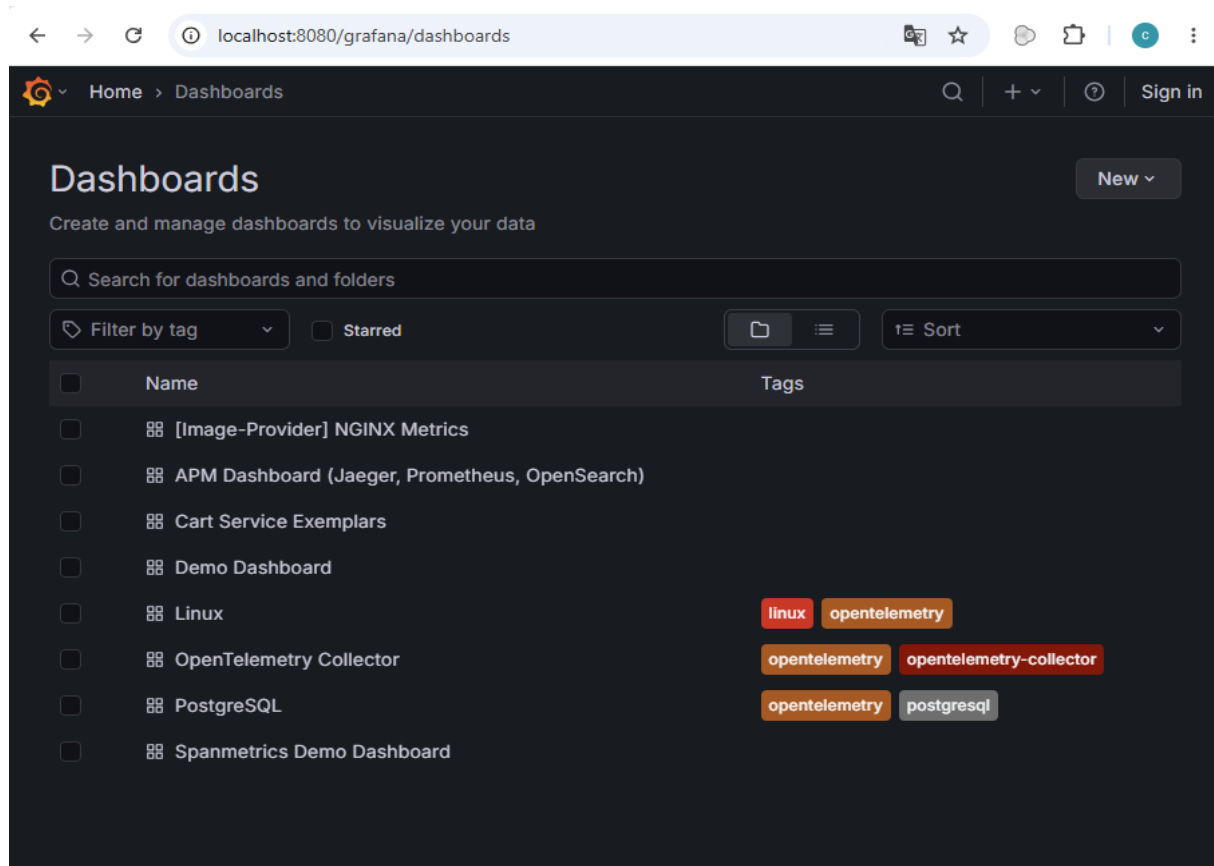


FIGURE 16 : LOCAL ACCESS TO THE APPLICATION'S WEB INTERFACES, JAEGER, AND GRAFANA

Exporting data to a Data Lake (MinIO / S3)

As mentioned in the introduction, the OpenTelemetry Collector architecture is modular and natively integrates multiple external exporters. Among them, the `awss3` [AWS S3 Exporter] allows telemetry to be sent directly to object storage systems compatible with the Amazon S3 API, such as MinIO [MinIO].

The integration of this external database, hosted on the SLICES-RI research infrastructure, could potentially serve as the first phase of solutions to two challenges in our experimental phase:

- *Reducing the hardware footprint.* Offloading storage and analysis from the worker nodes to avoid skewing the cluster's performance metrics.
- *Create a DataLake per node.* Use this persistent object storage to accumulate and archive observability signals from each machine separately. This will allow us to build datasets labeled by node, recording metrics and traces during various simulated attack scenarios, with the ultimate goal of training our anomaly detection algorithms.

To integrate this new exporter into our application, we created a new configuration manifest, `add_aws.yaml`. This file (Figure 17) declares the authentication environment variables and configures the endpoint to the SLICES-RI MinIO server.

Consequently, all trace and metric data received by the OpenTelemetry Collector is instantly transmitted to MinIO storage. We also have the option to configure batch processing to control the data flow—and thus the network load—using the batch parameter to be set in the S3uploader section (Figure 17).

```
opentelemetry-collector:
  extraEnvs:
    - name: AWS_ACCESS_KEY_ID
      value: "xxxxxx"
    - name: AWS_SECRET_ACCESS_KEY
      value: "xxxxx"

  config:
    exporters:
      awss3:
        s3uploader:
          region: "us-east-1"
          s3_bucket: "ilabt.imec.be-project-cloud-resilience"
          endpoint: "[https://s3.slices-be.eu](https://s3.slices-be.eu)"
          s3_force_path_style: true
          s3_prefix: "otel-data"
          marshaler: otlp_json # Formatage des données en JSON
          # ... (autres exportateurs)

    service:
      pipelines:
        metrics:
          exporters: [otlphttp/prometheus, debug, awss3]
        traces:
          exporters: [otlp, debug, spanmetrics, awss3]
```

FIGURE 17 : EXCERPT FROM THE ADD_AWS.YAML MANIFEST CONFIGURING THE AWSS3 EXPORTER AND ADDING IT TO THE METRICS AND TRACES PIPELINES.

To inject this file into the running configuration without interrupting the application, we use the Helm [Helm] update feature with the following command on the master node (node1):

```
sudo helm upgrade --install my-otel-demo open-telemetry/opentelemetry-demo --values
add_aws.yaml
```

The --upgrade --install argument combined with the --values flag instructs Kubernetes to merge our new manifest with the existing configuration. The Control Plane then dynamically restarts the OpenTelemetry Collector Pods so that they take this new export pipeline into account.

To validate the integration, we log in to the [SLICES-RI MinIO] web interface of the MinIO database provided by SLICES-RI (Figure 18).

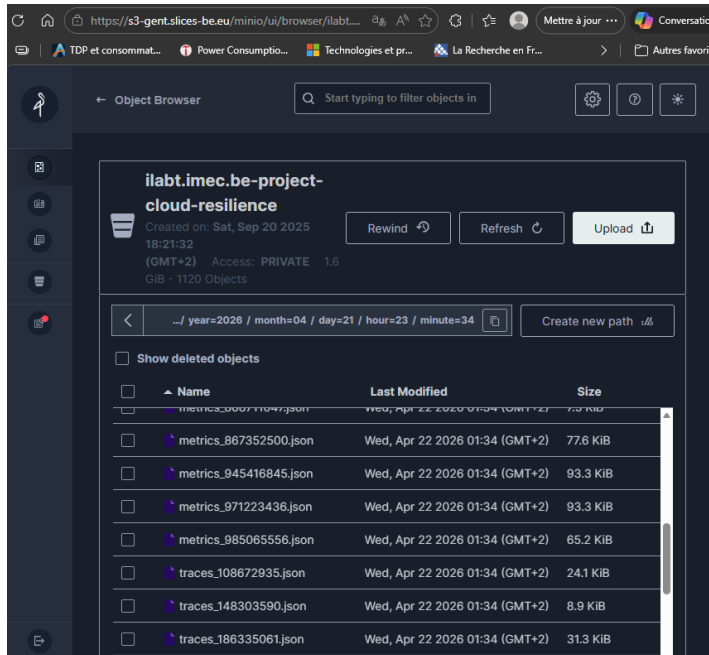


FIGURE 18 : MINIO DATABASE WEB INTERFACE.

Here we can see the automatic partitioning by date and time, as well as the actual storage of the streams in the form of `metrics_*.json` and `traces_*.json` files.

• ***Chaos Engineering and Chaos Mesh Architecture***

Chaos Mesh [Chaos Mesh] is a cloud-native, open-source chaos engineering platform that involves intentionally and controllably injecting failures—such as network outages, container crashes, or CPU saturation—into a distributed system to observe its behavior under stress.

In our use case, the integration of Chaos Mesh serves a dual purpose.

First, the tool will serve as the foundation for building a labeled dataset. By generating reproducible attacks and anomalies such as network latency or packet corruption, we can collect the altered telemetry via the OpenTelemetry Collector. This data will be essential for training the anomaly detection algorithm that will be implemented later.

Second, it highlights the limitations of standard monitoring. The injection of complex failures empirically demonstrates that passive observability approaches—such as simply visualizing spans or metrics in Jaeger and Grafana—reach their limits when it comes to identifying the root cause of a cascading failure. In a highly dynamic microservices environment, the volume of data generated by a single network error makes manual and visual analysis of the root cause nearly impossible without automated analytical tools [Anders, 2024].

Internal operation and injection mechanism in Kubernetes

The claim that Chaos Mesh injects faults without modifying application code is based on its integration with the Kubernetes API and Linux kernel primitives. Its architecture is built around several major components, including the Control Plane, the user interface, and the node agents [Chaos Mesh].

The Chaos Controller Manager serves as the brain of the Control Plane. This component extends the Kubernetes API through the use of CRDs (Custom Resource Definitions), a mechanism that teaches Kubernetes to understand new commands specific to chaos engineering. Although it is possible to issue imperative commands via YAML manifests, we will primarily use its built-in web interface called the Chaos Dashboard. This graphical interface allows you to configure and monitor attack scenarios visually while translating user actions into native Kubernetes objects in the background.

The Chaos Daemon is the execution component. It is deployed as a DaemonSet, a native Kubernetes controller that ensures an exact copy of a specific pod is always running on every worker node of the physical or virtual infrastructure [K8s DaemonSet]. This is the component that technically executes the breach.

When a network disruption experiment is initiated by the Control Plane via the web interface, the Chaos Daemon located on the target node does not interact directly with the application. It uses the host system's administrative privileges to enter the Network Namespace (Figure 19), which corresponds to the virtual and strictly isolated network environment provided by the Linux kernel to each pod to separate it from the rest of the system [Linux namespaces].

Once inside this namespace, the Chaos Daemon manipulates the Linux Traffic Control (TC) utility or configures iptables rules to alter incoming or outgoing traffic by introducing latency, packet loss, or corruption directly on the targeted pod's virtual network interface [Hubert et al.]. Thus, the microservices application experiences the failure in a completely transparent manner, generating altered telemetry that will be captured by our OpenTelemetry agents.

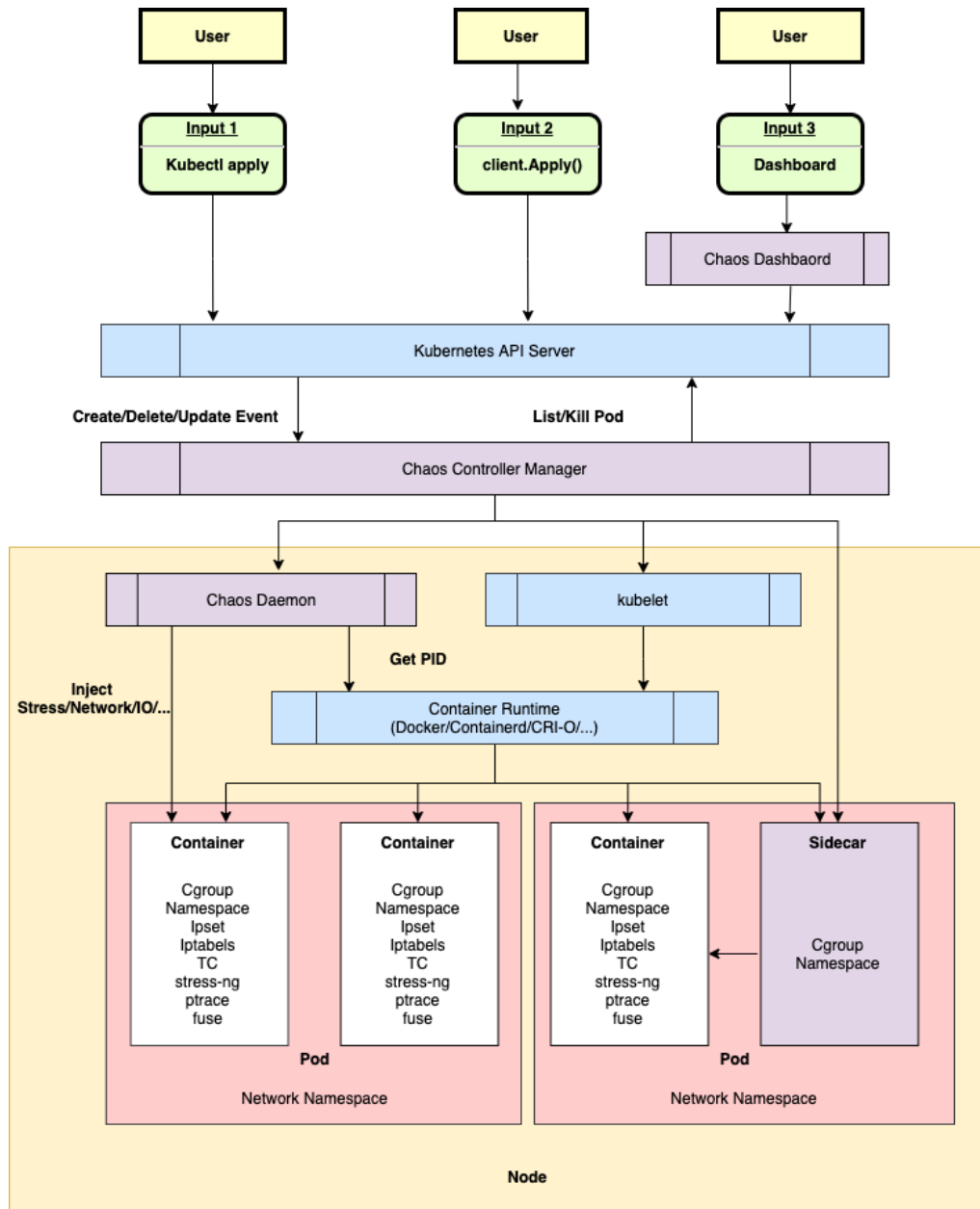


FIGURE 19 : OVERALL ARCHITECTURE AND CHAOS MESH INJECTION MECHANISMS [CHAOS MESH ARCH].

Figure 19 illustrates the execution flow from the user request via the Dashboard to the alteration of the container environment, where we can observe that the Chaos Daemon is capable of using various Linux utilities for different types of attacks, such as stress-ng for CPU saturation or ptrace for system calls, in addition to manipulating network primitives (TC, Ip-tables) within the Network Namespace.

Deploying Chaos Mesh on the cluster

The installation of the Chaos Mesh platform follows the same methodology as that of the otel-demo observability stack, using Helm.

The first step is to prepare the infrastructure to host the Control Plane components. We start by adding the official repository and creating a dedicated Namespace. In Kubernetes, a Namespace is a virtual environment that allows resources to be segmented and naming conflicts between applications to be avoided [K8s Namespaces].

```
# Add the official source repository and update the indexes
helm repo add chaos-mesh [https://charts.chaos-mesh.org](https://charts.chaos-mesh.org)
helm repo update
# Create the dedicated namespace
kubectl create ns chaos-mesh
```

The installation is performed using the helm install command. We inject specific parameters via the --set argument to adapt Chaos Mesh to the architecture of our nodes.

```
helm install chaos-mesh chaos-mesh/chaos-mesh \
  -n chaos-mesh \
  --set dashboard.create=true \
  --set chaosDaemon.runtime=containerd \
  --set chaosDaemon.socketPath=/run/containerd/containerd.sock
```

The --set parameters are used to ensure communication between the chaos-daemon and the hardware:

- dashboard.create=true: Enables the graphical management interface.
- chaosDaemon.runtime=containerd: Specifies that the container engine used on our VMs is containerd.
- chaosDaemon.socketPath: Defines the path to the socket, which is the container engine's physical communication endpoint. This allows the chaos-daemon to instruct the Linux kernel to act directly on the application's containers [Chaos Mesh Doc].

Once the pods are running, we need to access the Chaos Dashboard from our local machine. Since the service is deployed within the cluster's private network, we use the command:

```
kubectl port-forward -n chaos-mesh svc/chaos-dashboard 2333:2333
```

2333 is the default port assigned by the Chaos Mesh developers for its API and web interface. This port-forwarding mechanism creates a secure tunnel between port 2333 on our local machine and port 2333 on the Kubernetes service, enabling visual management of attacks.

As shown in Figure 20, which illustrates the different phases of the installation, the terminal output logs confirm the creation of resources and the "Running" status of the various components (Controller Manager and Daemons) within the chaos-mesh namespace.

```
ubuntu@node1:~$ sudo kubectl create ns chaos-mesh
namespace/chaos-mesh created
ubuntu@node1:~$ sudo helm install chaos-mesh chaos-mesh/chaos-mesh \
  -n chaos-mesh \
  --set dashboard.create=true \
  --set chaosDaemon.runtime=containerd \
  --set chaosDaemon.socketPath=/run/containerd/containerd.sock
NAME: chaos-mesh
LAST DEPLOYED: Mon May 11 13:30:36 2026
NAMESPACE: chaos-mesh
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
1. Make sure chaos-mesh components are running
   kubectl get pods --namespace chaos-mesh -l app.kubernetes.io/instance=chaos-mesh
ubuntu@node1:~$ sudo kubectl get pods --namespace chaos-mesh -l app.kubernetes.io/instance=chaos-mesh
NAME                                READY   STATUS    RESTARTS   AGE
chaos-controller-manager-9b7665fff-gm2ww  1/1     Running   0           26s
chaos-controller-manager-9b7665fff-n4bb5  1/1     Running   0           26s
chaos-controller-manager-9b7665fff-q99fh  1/1     Running   0           26s
chaos-daemon-4h9nk                      1/1     Running   0           26s
chaos-daemon-779mf                       1/1     Running   0           26s
chaos-daemon-fvlgv                       1/1     Running   0           26s
chaos-dashboard-5b77585c6c-hnx96         1/1     Running   0           26s
chaos-dns-server-6bccb5bb76-6zbxl        1/1     Running   0           26s
ubuntu@node1:~$ sudo kubectl port-forward -n chaos-mesh svc/chaos-dashboard 2333:2333
Forwarding from 127.0.0.1:2333 -> 2333
Forwarding from [::1]:2333 -> 2333
```

FIGURE 20 : PROCESS FOR CREATING AN AUTHENTICATION TOKEN

Once the components are up and running, access to the web interface is achieved from our local machine via the address localhost:2333. Upon arriving at the home page, the system blocks access and requires the entry of an authorization token.

This security requirement stems from Kubernetes' native architecture, which is based on the RBAC (Role-Based Access Control) mechanism [K8s RBAC]. Since Chaos Mesh has the ability to destroy or alter cluster resources, it does not allow any actions by default. To interact with the interface, the user must provide cryptographic proof in the form of a token certifying that they possess the required administrative rights on a specific Namespace.

The process of creating and injecting this identity is illustrated in 5 steps in Figure 21

Step 1 Authorization Request: The interface requires a token to validate the session. By clicking the Click here to generate wizard, the tool offers to configure a custom profile.

Step 2 Manifest generation: We explicitly target the otel-demo Namespace with a Manager privilege level. The wizard then generates a manifest named rabac.yaml consisting of three Kubernetes objects [K8s Service Accounts]

- A ServiceAccount: This is an identity that the Dashboard will use to authenticate with the Kubernetes API.

- A Role: A dictionary defining the authorized actions.
- A RoleBinding: The logical bridge that attaches these permissions to the ServiceAccount.

Step 3 Defining Custom Privileges: Inspecting the rbac.yaml file in the node terminal shows that this role limits our actions to the essentials. It allows only read access to pods, but grants modification rights (create, delete, patch) exclusively on the chaos-mesh.org API group [Chaos Mesh Permissions].

Steps 4 & 5 Application and extraction: After applying this configuration file with kubectl apply, we use the kubectl create token command to force Kubernetes to generate the JWT (JSON Web Token). It is this string generated in the terminal that, once copied into the Dashboard, unlocks access to the chaos creation platform.

The image illustrates the process of creating an authentication token for the Chaos Mesh dashboard. It is divided into three main sections:

- Dashboard Token Generator:** On the right, the 'Token generator' panel shows the 'Namespace' set to 'otel-demo' and the 'Role' set to 'Manager'. A yellow '1' highlights the role selection.
- Terminal Application:** The terminal at the bottom shows the command `kubectl apply -f rbac.yaml` being executed, followed by `kubectl create token account-otel-demo-manager-efxow`. A yellow '2' highlights the role selection in the dashboard.
- Token Extraction:** The terminal shows the command `kubectl describe -n otel-demo secrets account-otel-demo-manager-efxow` being used to retrieve the token. A yellow '3' highlights the token value in the terminal output.
- Dashboard Submission:** The modal on the left prompts for the token. A yellow '4' highlights the 'Submit' button.

FIGURE 21 : AUTHENTICATION TOKEN CREATION PROCESS

- **Trace Analysis**

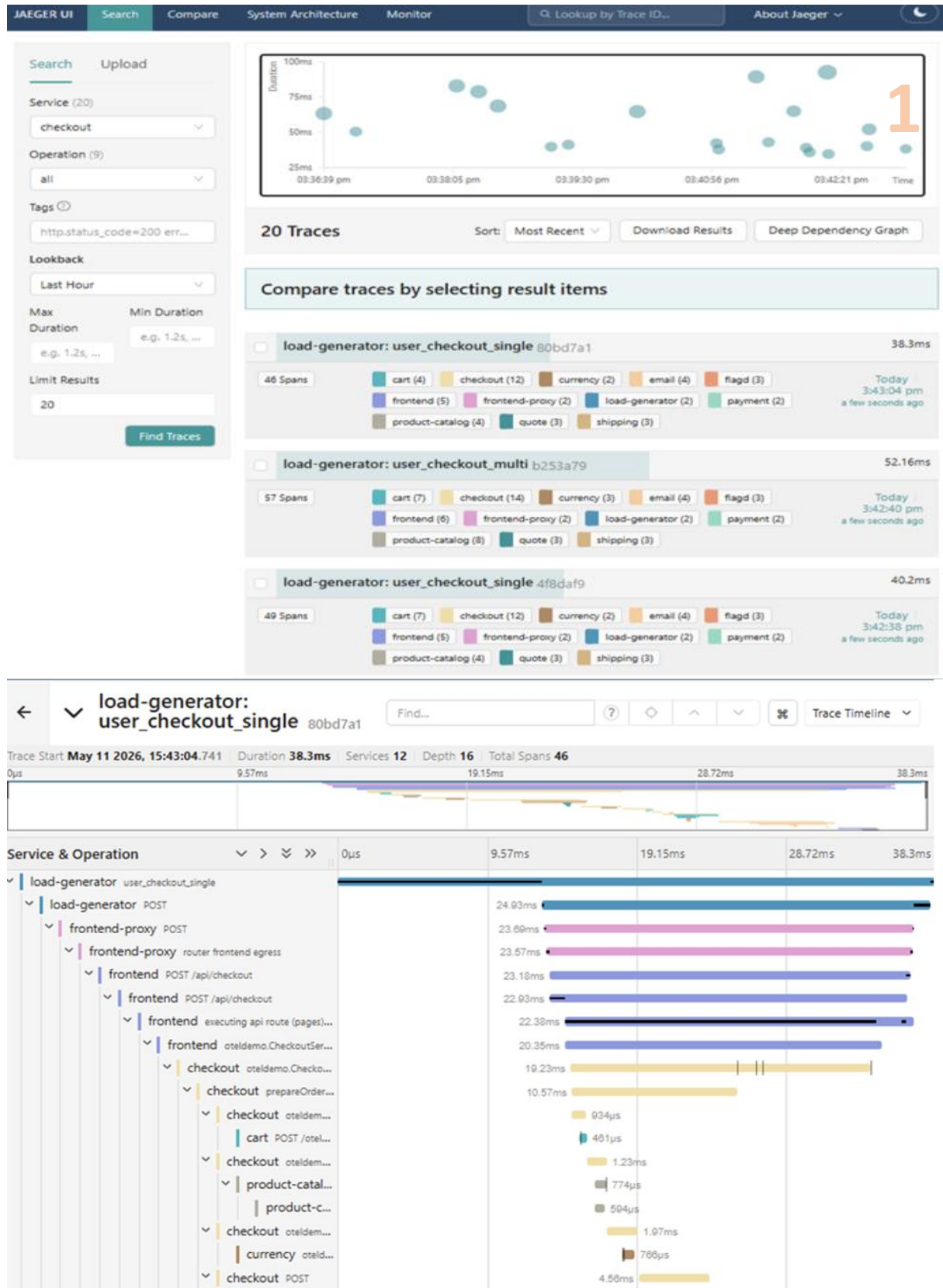
Before injecting anomalies into our cluster, it is important to understand how the application operates in its normal state. To do this, we use Jaeger to analyze traces through its locally accessible interface.

As previously established, the OpenTelemetry Collector extracts telemetry from the nodes and exports it to our various backends. When Jaeger receives this data via its ingestion port using the OTLP or gRPC protocol, it reconstructs the complete path of a user request across the entire microservices network [Jaeger Arch].

Jaeger's operation is based on two concepts standardized by the World Wide Web Consortium [W3C Trace Context]. The first is the Trace, which represents the complete lifecycle of a request, such as a user validating a shopping cart. It is identified by a unique identifier called a TraceID, generated as soon as the request enters the cluster. The second concept is the Span. It constitutes the basic unit of a trace and represents a specific operation within a microservice, such as a database query or a call to another service. Each span has its own identifier, a start timestamp, a precise duration, and, most importantly, the identifier of its parent to maintain the execution hierarchy.

It is the injection of these identifiers into request headers, via a context propagation mechanism, that allows Jaeger to link events occurring on different nodes to draw an execution graph [OTEL Tracing Concepts].

To illustrate this mechanism, we have isolated a typical request generated by the Locust load testing tool [Locust] regarding the payment process. Figure 22 is a composite view of our various analysis interfaces, ranging from global search to the inspection of metadata for a specific span.



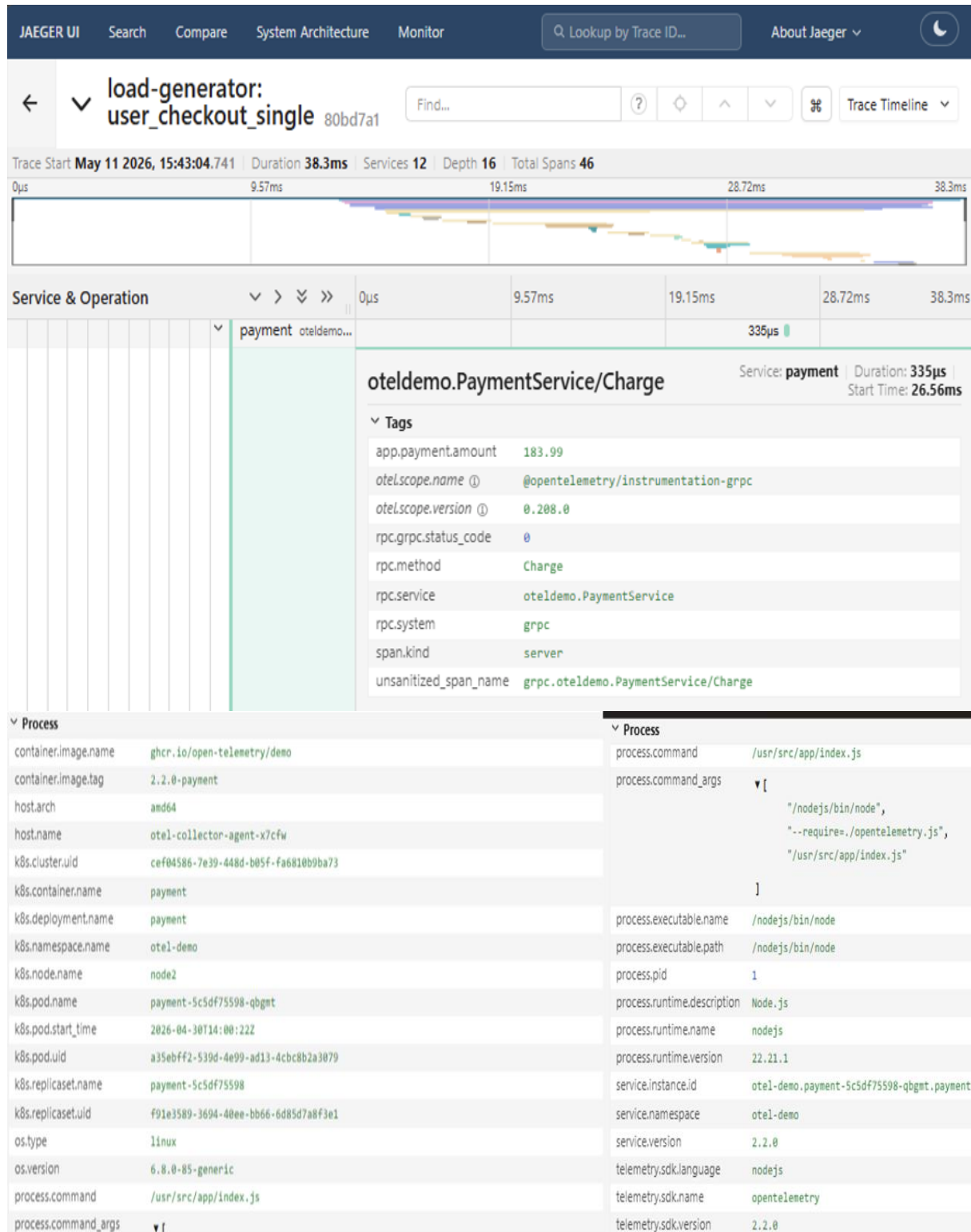


FIGURE 22 : JAEGER INTERFACE: HOME SCREEN, GANTT CHART, PAYMENT SPAN TAGS, AND THE PROCESS SECTION.

Step 1 presents Jaeger's global search interface. This view includes filtering options to target specific events. In our case, we filtered the results by the checkout service to isolate the purchase processes. The scatter plot visually displays the distribution of response times over a given period. From these results, we select a specific trace with a total duration of 38.3 milliseconds to inspect it in detail.

Step 2 displays the Gantt chart for the selected trace. We observe that the root trace begins on the frontend-proxy and then moves to the frontend with an HTTP POST request. Jaeger visually demonstrates that this service acts as the main entry point by triggering sub-requests to the checkout service via the gRPC protocol. The checkout service, in turn, sequentially calls the cart service to retrieve items, the catalog service, the currency service for conversion, the shipping service, and finally the payment service to validate the card transaction.

In this normal operation profile, we observe several key metrics, such as normal latency, which is confirmed by the overall execution time of the full trace at 38.3 ms. Internal calls between pods take on the order of milliseconds, as indicated by the duration of 335 microseconds for the specific payment operation.

Step 3 illustrates the options provided by Jaeger to inspect the payment span's metadata in detail. In the Tags section, we obtain the network context of the operation. The `rpc.system` attribute indicates the use of the gRPC protocol. The success of the transaction is evidenced by the `rpc.grpc.status_code` value set to 0, which corresponds to the standard success code in the gRPC architecture [gRPC Status Codes].

Finally, Step 4 focuses on the Process section and demonstrates the method for capturing this data. The presence of the `--require=./opentelemetry.js` runtime argument, coupled with the `telemetry.sdk.language` variable identified as `nodejs`, confirms the use of auto-instrumentation. It is this OpenTelemetry agent, dynamically injected into the container's runtime environment, that intercepts traffic at the source. Furthermore, the presence of Kubernetes attributes such as `k8s.node.name` pointing to `node2` or `k8s.namespace.name` is not generated by the application itself. This enrichment is performed downstream by the OpenTelemetry Collector's `k8sattributes` processor, which automatically adds cluster-related metadata to spans before they are exported to Jaeger [K8s Attributes Processor].

- **Extraction and analysis in Python from the MinIO database**

Beyond visual inspection via the Jaeger interface, we can also analyze this raw data by developing a Python script (Figure 24) that connects directly to our database.

Communication with the storage component is handled by importing the Boto3 library, which is the standard software development kit for interacting with Amazon S3-compatible programming interfaces [AWS Boto3].

This extraction approach offers an advantage when processing large datasets. Loading large volumes of traces and metrics directly into RAM could slow down the machine running the analysis or even cause it to crash due to overload. To work around this issue, our script does not pull the entire database. Instead, it leverages the time-based partitioning generated by the OpenTelemetry Collector by targeting a specific prefix- e corresponding to the year, month, day, and minute of the experiment. The script then downloads and deserializes the JSON files sequentially, processing one batch before moving on to the next.

To validate the effectiveness of this approach, we instrumented our code with Tracemalloc, a native module of the Python standard library designed to trace memory allocation and identify consumption spikes [Python tracemalloc].

Figure 23 illustrates the capabilities of this script through three separate runs merged for analysis.

```
ubuntu@pc02:~/Desktop$ python3 extract_data_.py
[INFO] Connecting to MinIO Data Lake...

[INFO] --- ANALYZING 5 TRACE FILES ---
[TRACE] TraceID: 17d0b... | Service: load-generator | Duration: 25ms | Status: OK
[TRACE] TraceID: 108ad... | Service: load-generator | Duration: 19ms | Status: OK
[TRACE] TraceID: 1d19a... | Service: load-generator | Duration: 8ms | Status: OK
[TRACE] TraceID: 79b98... | Service: load-generator | Duration: 1ms | Status: OK
[TRACE] TraceID: 17d0b... | Service: load-generator | Duration: 26ms | Status: OK
[INFO] Total: 915 transactions analyzed for the period of 2026-04-21 at 23:33.

[INFO] --- ANALYZING 5 METRIC FILES ---
[METRIC] Service: recommendation | Name: system.cpu.time | Value: 775.34 seconds
[METRIC] Service: recommendation | Name: system.cpu.utilization | Value: 0.819 (ratio / counter)
[METRIC] Service: recommendation | Name: system.memory.usage | Value: 4165169152 Bytes
[METRIC] Service: recommendation | Name: system.memory.utilization | Value: 0.5015 (ratio / counter)
[METRIC] Service: recommendation | Name: process.cpu.time | Value: 3.08 seconds
[INFO] Total: 1170 metric points analyzed for the period of 2026-04-21 at 23:33.

[SYSTEM] Maximum script RAM consumption: 12.7 MB
ubuntu@pc02:~/Desktop$ python3 extract_data_.py

[SEARCH] Searching for span for service: payment

=====
SERVICE & OPERATION : payment / charge
=====
Trace ID : 4c3e10d696c1d48c76d894e308536ddf
Span ID : 7e5f52c63aec159b
Parent ID : 946ac48789261451
Duration : 0.34 ms

--- TAGS ---
app.loyalty.level : bronze
app.payment.card_type : visa
app.payment.card_valid : True
app.payment.charged : True

--- PROCESS ---
host.arch : amd64
host.name : otel-collector-agent-j8n5l
k8s.deployment.name : payment
k8s.namespace.name : otel-demo
k8s.node.name : node5
k8s.pod.name : payment-5c5d475598-hwz45
k8s.pod.start_time : 2026-04-21T23:00:58Z
k8s.pod.uid : b8568194-cfe4-4ac2-9389-4c9d5d85e983
os.type : linux
os.version : 6.8.0-85-generic
process.command : /usr/src/app/index.js
process.command_args : {'arrayValue': {'values': [{'stringValue': '/nodejs/bin/node'}, {'stringValue': '--require=./opentelemetry.js'}, {'stringValue': '/usr/src/app/index.js'}]}}
process.executable.name : /nodejs/bin/node
process.executable.path : /nodejs/bin/node
process.pid : 1
process.runtime.description : Node.js
process.runtime.name : nodejs
process.runtime.version : 22.21.1
service.instance.id : b8568194-cfe4-4ac2-9389-4c9d5d85e983
service.name : payment
service.namespace : otel-demo
telemetry.sdk.language : nodejs
telemetry.sdk.name : opentelemetry
telemetry.sdk.version : 2.2.0
=====
```

```
ubuntu@pc02:~/Desktop$ python3 extract_data_.py
SEARCH trace ID : 4c3e10d696c1d48c76d894e308536ddf

[SUCCESS] Trace 4c3e10d696c1d48c76d894e308536ddf found
{
  "traceId": "4c3e10d696c1d48c76d894e308536ddf",
  "spanId": "59865f2b163a1291",
  "parentSpanId": "0bccafc994574963",
  "flags": 769,
  "name": "POST /oteldemo.CartService/GetCart",
  "kind": 2,
  "startTimeUnixNano": "1776814411315742800",
  "endTimeUnixNano": "1776814411316434900",
  "attributes": [
    {
      "key": "server.address",
      "value": {
        "stringValue": "cart"
      }
    },
    {
      "key": "server.port",
      "value": {
        "intValue": "8080"
      }
    },
    {
      "key": "http.request.method",
      "value": {
        "stringValue": "POST"
      }
    },
    {
      "key": "url.scheme",
      "value": {
        "stringValue": "http"
      }
    },
    {
      "key": "url.path",
      "value": {
        "stringValue": "/oteldemo.CartService/GetCart"
      }
    },
    {
      "key": "network.protocol.version",
      "value": {
        "stringValue": "2"
      }
    }
  ]
}
```

FIGURE 23 : SCREENSHOT OF THE OVERALL ANALYSIS AND A DETAILED TRACE OF THE PAYMENT SERVICE

The first part of the figure shows the overall execution of the script. The tool identifies and analyzes the files present in the target directory. It successfully extracts information from 915 transaction traces and filters 1,170 metrics specific to CPU and memory consumption. Despite the volume of data processed, the script's maximum RAM consumption reaches only 12.7 MB.

The second part of the figure extracts specific data from the set of traces. In this specific case, the goal is to isolate a span corresponding to the payment service. The script parses the OTLP format to accurately retrieve the trace ID, the exact duration of 0.34 milliseconds, and all metadata-related dictionaries. The values obtained in the terminal, such as the card approval status or the Kubernetes pod ID, are identical to those in the Jaeger interface.

The third part of the figure displays the raw JSON output of a complete span. In this example, the script targeted a specific trace ID associated with the shopping cart service named CartService. It includes timestamps, hierarchy identifiers such as spanId and parentSpanId, as well as the array of attributes detailing the network request.

Algorithm : Data Extraction and Programmable Profiling

Input: Raw telemetry data stored in MinIO within the detection time window T , target service object S_{target} , target trace identifier id_{target} , and threshold limit K .

Output: Extracted traces T_{ext} , aggregated metrics M , detailed span graph node D , raw payload J , and peak memory R_{peak} .

```

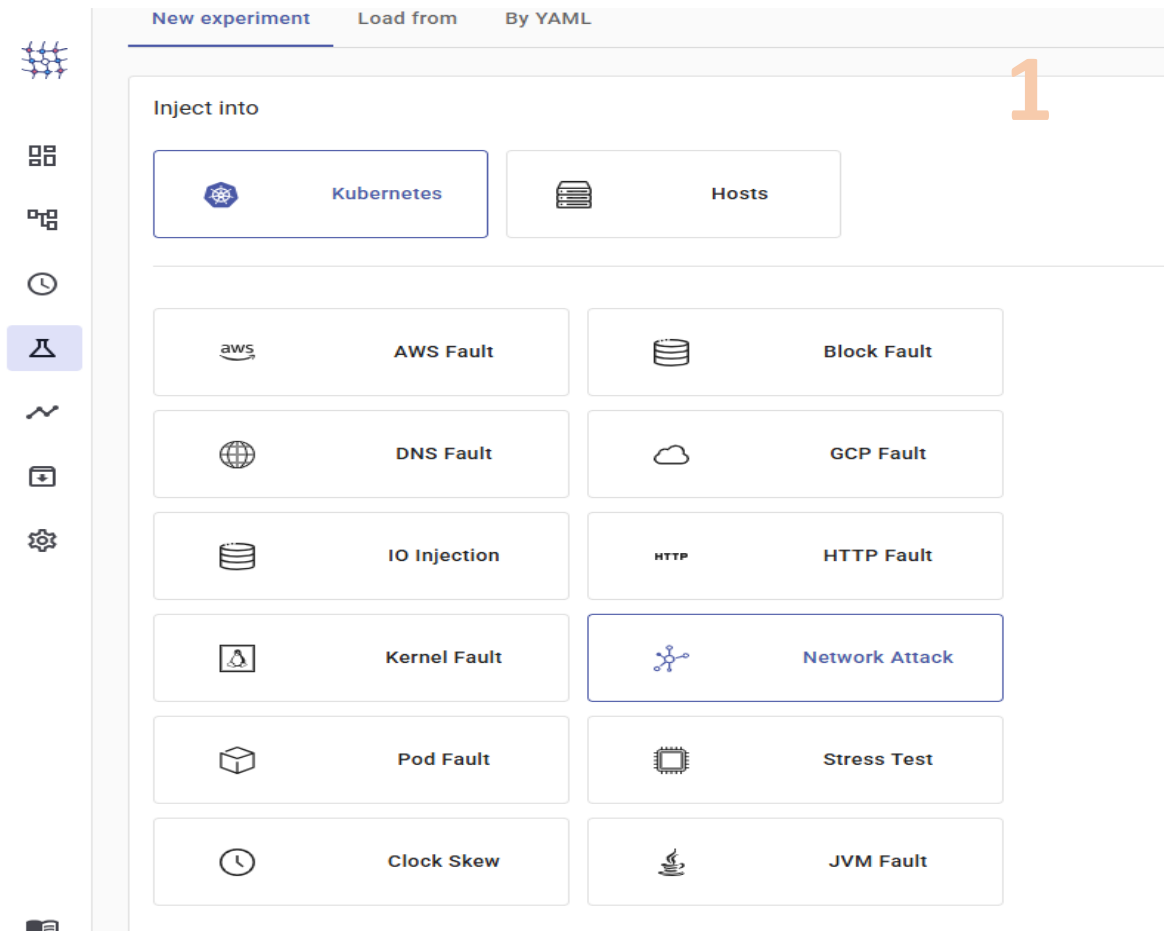
1 The Data Pre-process Module initializes the S3 client connection and starts the tracemalloc memory tracking module.
2 Retrieve the list of objects  $F$  from the bucket matching the temporal prefix  $T$ .
3 Partition  $F$  into the trace dataset  $F_{trace}$  and the metric dataset  $F_{metric}$ .
4 The Global Analysis Module iterates over  $F_{trace}$  and  $F_{metric}$ .
5 Parse the OTLP JSON structure to extract source properties, calculate span duration  $\Delta t$ , and verify execution status.
6 Append the parsed entities to  $T_{ext}$  and  $M$  until the threshold limit  $K$  is reached.
7 The Targeted Inspection Module traverses  $F_{trace}$  sequentially to locate  $S_{target}$ .
8 if service.name equals  $S_{target}$  then
9     Extract operational tags  $A$  and structural process metadata  $P$ .
10    Construct the detailed node representation  $D$  and terminate the search loop.
11 end
12 The Payload Extraction Module traverses  $F_{trace}$  to pinpoint  $id_{target}$ .
13 if traceId equals  $id_{target}$  then
14    Isolate the exact JSON payload  $J$  and terminate the search loop.
15 end
16 Stop the memory tracking module and record the maximum RAM consumption  $R_{peak}$ .

```

FIGURE 24 : HOW THE PYTHON CODE WORKS

- **Configuring the network attack via the Chaos Mesh dashboard**

For controlled fault injection, the Chaos Mesh graphical interface is used. A new experiment is created through a process accessible from the main main menu. Figure 25 details this configuration.



2










Latency

2000ms

The latency of delay

Jitter

The jitter of delay

Correlation

The correlation of delay

Direction

to

Specify the network direction

External targets

Type and end with Enter to generate the network targets outside k8s

Target

Submit

The screenshot displays the E2CC interface for configuring and executing a network attack experiment. The interface is divided into two identical panels, each with a large orange number (3 and 4) indicating the steps.

Panel 3 (Top):

- Inject into:** Status is 'Inject into' with a green checkmark.
- Experiment Info:**
 - Scope:**
 - Namespace Selectors:** `otel-demo` (Specifies the namespaces to which target Pods belong.)
 - Label Selectors:** `app.kubernetes.io/component: payment` (Specifies the labels that target Pods come with.)
 - Mode:** `All` (Specifies the mode of the experiment.)
 - Preview of Pods to be injected:** Checking or unchecking Pods to further limit the scope.
- Metadata:**
 - Name:** `otel-demo` (Fill the experiment name.)
 - Run:** `Run continuously` (Toggle switch is on).
 - Duration:** `5m` (Supported formats of the duration are: ms / s / m / h.)
- Submit:** Button to execute the experiment.

Panel 4 (Bottom):

- Inject into:** Status is 'Inject into' with a green checkmark.
- Experiment Info:**
 - Scope:**
 - Namespace Selectors:** `otel-demo` (Specifies the namespaces to which target Pods belong.)
 - Label Selectors:** `app.kubernetes.io/component: payment` (Specifies the labels that target Pods come with.)
 - Mode:** `All` (Specifies the mode of the experiment.)
 - Preview of Pods to be injected:** Checking or unchecking Pods to further limit the scope.
- Metadata:**
 - Name:** `otel-demo` (Fill the experiment name.)
 - Run:** `Run continuously` (Toggle switch is on).
 - Duration:** `5m` (Supported formats of the duration are: ms / s / m / h.)
- Submit:** Button to execute the experiment.

FIGURE 25 : CONFIGURING AND EXECUTING THE NETWORK ATTACK WITH A 2-SECOND LATENCY INJECTION

Selection begins with the anomaly category. In the New Experiment menu, the Kubernetes system is specified as the target environment. Among the various failure scenarios offered by the platform, the NetworkAttack option is selected.

The next phase involves configuring the disruption parameters. An artificial latency of 2000 ms is set. The temporal variation and correlation parameters are ignored to ensure a fixed and predictable delay during analysis. The traffic direction is also specified to target incoming packets.

Defining the scope of action involves applying filters to restrict this latency rule exclusively to pods associated with the payment service, located within the application namespace. A continuous runtime of 5 minutes is also set for our test case.

Finally, submitting the form confirms that the attack was executed successfully. The interface displays the history of events related to the experiment (Figure 26). The system logs confirm that the Chaos Daemon component successfully applied the traffic control rules to the targeted pod's virtual network interface. The experiment status changes to Running, indicating that latency is now active on the infrastructure.

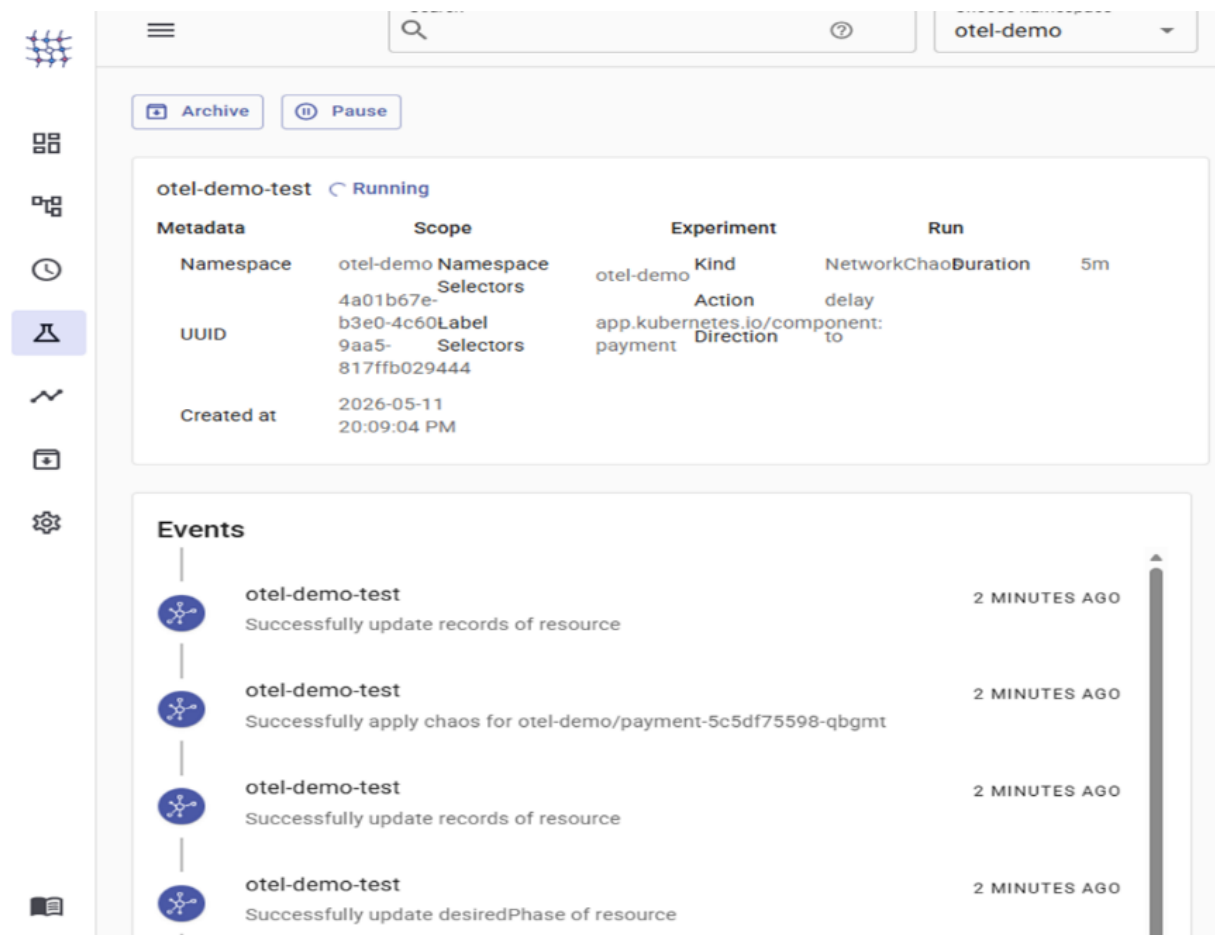


FIGURE 26 : EXECUTION OF THE NETWORKCHAOS ATTACK

To visualize the results of this injection, a review of the Jaeger interface allows for a comparison of trace data before and during the attack. Figure 27 highlights the direct impact of the network disruption on the overall architecture.

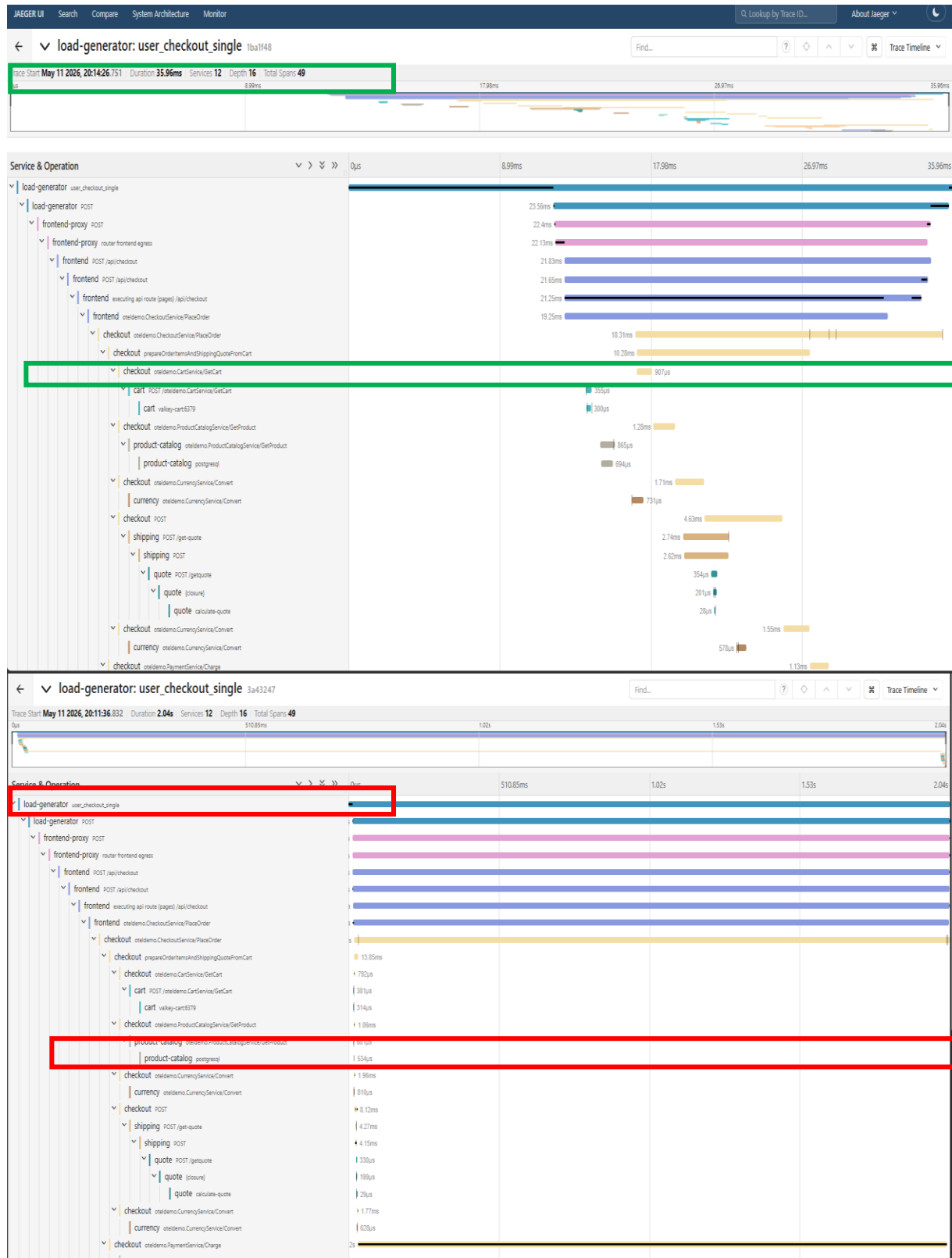


FIGURE 27 : RESULT OF THE CHAOS-MESH EXECUTION

The first part of the figure shows the Gantt chart of a normal transaction. The overall execution of the purchase process completes in 35.96 ms.

The second part shows the same transaction captured during the Chaos Mesh attack. The disruption in the execution timeline is evident, with the total trace duration reaching 2.04 seconds. A detailed inspection of the execution segments identifies the source of the slowdown. The bar associated with the payment service stretches out and, on its own, accounts for the entire 2-second latency configured in the previous step.

This comparison illustrates the phenomenon of cascading failures in microservices-based architectures. Since the payment service is experiencing network slowdown, the parent service responsible for order validation is forced to wait for the gRPC request to return. Due to reverse propagation, the ingress proxy and the application frontend service experience a nearly identical wait. A network disruption targeting a single, isolated component is thus sufficient to degrade the overall response time for the end user.

To demonstrate the limitations of tracing tools like Jaeger, a new experiment is launched. This time, the injection involves a CPU stress test on the cartservice, a component that performs multiple internal calculations. Figure 28 details the configuration of this attack.

CPU

Workers

3

CPU workers

Load

100

CPU load

Options of CPU stressors

Type and end with Enter to generate the stress-ng options

Memory

Workers

0

Memory workers

Size

Memory size specifies the memory size to be occupied or a percentage of the total memory size

Options of Memory stressors

Type and end with Enter to generate the memStress options

More Options

Submit

Experiment Info

Scope

Namespace Selectors

otel-demo

Specifies the namespaces to which target Pods belong.

Label Selectors

app.kubernetes.io/component: cart

Specifies the labels that target Pods come with.

More Options

Mode

All

Specifies the mode of the experiment.

Preview of Pods to be injected

Checking or unchecking Pods to further limit the scope.

	Name	Namespace	IP address
☒	cart-74f5bf48d9-5xbc7	otel-demo	10.2

Metadata

Name

otel-demo-test4

Fill the experiment name.

More Options

Run

☐ Run continuously

Duration

10m

Supported formats of the duration are: ms / s / m / h.

Submit

otel-demo-test4 Running

Metadata	Scope	Experiment	Run
Namespace	otel-demo	Kind	StressChaos
UUID	e7667bd2-5d6d-4578-a69c-272868cb7861	CPU	workers: 3 size:
Created at	2026-05-12 13:22:10 PM	Label Selectors	Duration
		app.kubernetes.io/component: cart	10m

Events

otel-demo-test4

Successfully update records of resource

4 MINUTES AGO

otel-demo-test4

Successfully apply chaos for otel-demo/cart-74f5bf48d9-5xbc7/cart

4 MINUTES AGO

otel-demo-test4

Successfully update desiredPhase of resource

4 MINUTES AGO

otel-demo-test4

Experiment has started

4 MINUTES AGO

otel-demo-test4

Successfully update finalizer of resource

4 MINUTES AGO

otel-demo-test4

Finalizer has been deleted

4 MINUTES AGO

Definition

```
1 kind: StressChaos
2 aptVersion: chaos-mesh.org/v1alpha1
3 metadata:
4   namespace: otel-demo
5   name: otel-demo-test4
6 spec:
7   selector:
8     namespaces:
9     - otel-demo
10    labelSelectors:
11      app.kubernetes.io/component: cart
12 mode: all
13 stressors:
14   cpu:
15     workers: 3
16     load: 100
17 duration: 10m
18
```

Download

FIGURE 28 : STRESSCHAOS CONFIGURATION AND EXECUTION

A StressChaos-type scenario is defined with 3 out of 4 processes running simultaneously, configured to saturate the CPU load at 100%. The scope of the test is limited to pods labeled with the cart service running on node 4, for a continuous execution duration of 10 minutes.

Inspection of the results in Jaeger during the attack reveals counterintuitive application behavior that is fundamental to this demonstration. Figure 29 shows the trace captured during the saturation.

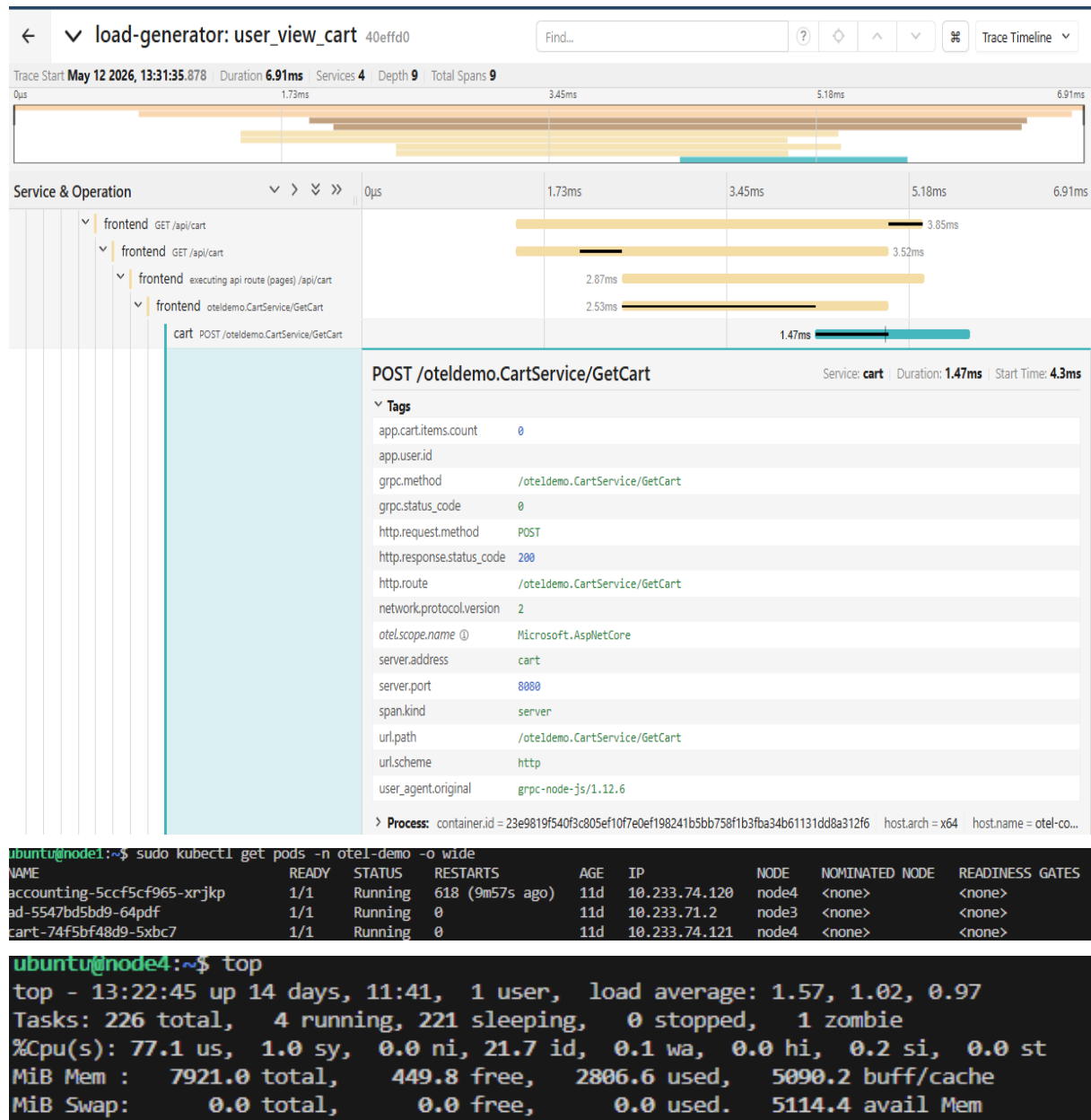


FIGURE 29 : CAPTURE OF THE CARTSERVICE JAEGER TRACE AT 1.47 MS AND THE STATE OF NODE4

As shown in Figure 29, the cart service targeted by the experiment is running on node4. Direct analysis of this node via the top command confirms heavy CPU saturation with an application load of 77.1%. Despite this CPU saturation confirmed by the system and the events

on the Chaos Mesh dashboard, the captured trace shows a short execution time of 1.47 ms for the operation. The diagram's structure remains visually sound, and no latency is observable. Furthermore, examining the metadata in the Tags segment indicates an HTTP 200 success status and the absence of error codes. Most importantly, the section related to the process does not report any error information.

This highlights that traditional monitoring tools like Jaeger, or those based on threshold rules, have limitations when it comes to investigating infrastructure anomalies. These shortcomings revolve around:

- The inability to identify silent gray failures in the absence of explicit error codes reported by the application [Huang et al., 2017].
- The lack of visibility into the state of the infrastructure, as distributed traces capture the execution flow but ignore contention for underlying hardware resources [Zhou et al., 2021].
- The inefficiency of manually or visually detecting micro-variations in latency when they are buried in the massive volume of cloud-native network traffic [Nedelkoski et al., 2019].
- The inability to natively correlate application execution traces with purely hardware-based events derived from metrics without an external cross-analysis system [Jiang et al., 2023].

4.1.4 Security

- **Cascading Detection**

A first contribution in terms of security involves studying detection mechanisms specifically tailored for runtime execution, capable of providing rapid decisions while limiting the load imposed on the infrastructure. Unlike much of the literature, which is evaluated solely offline, the goal is to account for the constraints of cloud environments, particularly detection time, resource consumption, and scalability.

With this in mind, a cascading detection strategy can be considered. The main idea is to break down the detection process into several stages of increasing complexity. A first lightweight layer performs rapid data filtering to identify unlikely behaviors. This stage deliberately favors low-cost methods that do not require deep learning models. The goal is to quickly reduce the volume of data requiring in-depth analysis.

Events detected as suspicious are then forwarded to a second, more advanced detection layer capable of performing a more detailed analysis of the observed behavior. This separation allows the most resource-intensive processing to be reserved for a smaller subset of data, thereby limiting the overall impact on system performance.

4.1.5 Conclusion

- **Summary of the Work Performed**

This report has demonstrated how to design, deploy, and evaluate a complete telemetry pipeline tailored to the constraints of distributed microservice architectures. Through the deployment of the experimental platform with the otel-demo microservices application and the use of chaos engineering via Chaos Mesh to simulate controlled failures, we have highlighted the role of trace and metric collection in understanding a system's dynamic behavior. Above all, this approach has demonstrated the inherent limitations of traditional monitoring tools. In the face of gray failures or silent hardware saturations, visual analysis or analysis based on static thresholds proves insufficient to isolate the root cause. Furthermore, we addressed the challenge of hardware overload on constrained execution nodes by asynchronously offloading this data to persistent storage such as MinIO.

- **Research and Integration Prospects**

This massive, centralized data collection serves as the starting point for our future work, which will aim to address the challenge of manually correlating application and hardware events. Since a distributed trace can be structurally modeled as a directed acyclic graph (DAG), our goal is to merge these execution topologies with time series derived from metrics. By extracting specific attributes capable of characterizing anomalies, we plan to represent this data as heterogeneous graphs. This modeling will serve as the basis for training algorithms based on Graph Neural Networks (GNNs), which are particularly well-suited for capturing the spatial and temporal dependencies of microservices.

Ultimately, to reconcile this anomaly detection with the resource constraints and privacy challenges specific to Edge environments, the deployment of these models will adopt a federated learning approach, enabling knowledge to be extracted as close to the source as possible in a decentralized manner.

Finally, since all of this work has been validated on the SLICES-RI environment, one of our short-term objectives is to ensure the agnosticism and portability of these experiments. This will involve consolidating the deployment scripts to enable a seamless migration of this observability platform to the project's internal infrastructure.

Bibliography

[Anders, 2024] Anders, N. (2024). Tracing Overhead in Microservices Architectures.

[AWS Boto3] Amazon Web Services. Boto3 Documentation - The AWS SDK for Python. URL: <https://boto3.amazonaws.com/v1/documentation/api/latest/index.html>

[AWS S3 Exporter] OpenTelemetry. AWS S3 Exporter for OpenTelemetry Collector. URL: <https://github.com/open-telemetry/opentelemetry-collector-contrib/tree/main/exporter/awss3exporter>

[Beahan et al., 2025] Beahan, S., Ullah, F., Chalmers, L., Fatima, U., & Shahin, M. (2025). Characterizing Vulnerabilities in Microservices: Source, Age, and Severity.

[Chang et al., 2019] Chang, H., Kodialam, M., Lakshman, T., & Mukherjee, S. (2019). Micro-service Fingerprinting and Classification using Machine Learning. In 2019 IEEE 27th International Conference on Network Protocols (ICNP) (pp. 1–11). IEEE. DOI: 10.1109/ICNP.2019.8888077

[Chaos Mesh] Chaos Mesh. A Powerful Chaos Engineering Platform for Kubernetes. Cloud Native Computing Foundation (CNCF). URL: <https://chaos-mesh.org/docs/>

[Chaos Mesh Arch] Chaos Mesh. Chaos Mesh Architecture. URL: <https://chaos-mesh.org/assets/images/architecture-76301820de324f79d79db310b11b9246.png>

[Chaos Mesh Doc] Chaos Mesh. Configure Container Runtime. URL: <https://chaos-mesh.org/docs/production-installation-using-helm/#configure-container-runtime>

[Chaos Mesh Permissions] Chaos Mesh. Manage User Permissions. URL: <https://chaos-mesh.org/docs/manage-user-permissions/>

[Envoy Proxy] Envoy Project Authors. Envoy Proxy. Official CNCF Documentation. URL: <https://www.envoyproxy.io/>

[Grafana Datasources] Grafana Labs. Grafana Datasources Provisioning Configuration. URL: <https://github.com/open-telemetry/opentelemetry-demo/tree/main/src/grafana/provisioning/datasources>

[Grafana Labs] Grafana Labs. Grafana: The open observability platform.

[gRPC Status Codes] gRPC. gRPC Core: Status codes and their use in gRPC. URL: https://grpc.github.io/grpc/core/md_doc_statuscodes.html

[Hannousse & Yahiouche, 2021] Hannousse, A., & Yahiouche, S. (2021). Securing micro-services and microservice architectures: A systematic mapping study. Computer Science Review, vol. 41, 100415. DOI: 10.1016/j.cosrev.2021.100415

[Helm] Helm. The package manager for Kubernetes. Official CNCF documentation. URL: <https://helm.sh/docs/>

[Hizal et al., 2021] Hizal, S., Cavusoglu, U., & Akgun, D. (2021). A new Deep Learning Based Intrusion Detection System for Cloud Security. In 2021 3rd International Congress on Human-Computer Interaction, Optimization and Robotic Applications (HORA) (pp. 1–4). IEEE. DOI: 10.1109/HORA52670.2021.9461285

[Huang et al., 2017] Huang, P., Guo, C., Lorch, J. R., Zhou, L., & Dang, Y. (2017). Gray Failure: The Achilles' Heel of Cloud-Scale Systems. In Proceedings of the 16th ACM Workshop on Hot Topics in Networks (HotNets). DOI: 10.1145/3152434.3152436

[Hubert et al.] Hubert, B., et al. Linux Advanced Routing & Traffic Control HOWTO (LARTC), Queueing Disciplines for Bandwidth Management (tc). The Linux Documentation Project. URL: <https://tldp.org/HOWTO/Adv-Routing-HOWTO/lartc.qdisc.html>

[Jaeger] Jaeger. Open source, end-to-end distributed tracing. URL: <https://www.jaegertracing.io/>

[Jaeger Arch] Jaeger. Jaeger Architecture and Components. URL: <https://www.jaegertracing.io/docs/latest/architecture/>

[Jiang et al., 2023] Jiang, Z., Liu, J., Huang, J., Huo, Y., Gu, J., et al. (2023). Nezha: Interpretable Fine-Grained Root Cause Analysis for Microservices on Multi-modal Observability Data. In 2023 USENIX Annual Technical Conference (USENIX ATC 23) (pp. 633–648).

[K8s Attributes Processor] OpenTelemetry. Kubernetes Attributes Processor. URL: <https://github.com/open-telemetry/opentelemetry-collector-contrib/tree/main/processor/k8sattributesprocessor>

[K8s DaemonSet] Kubernetes. DaemonSet. Kubernetes Official Documentation. URL: <https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/>

[K8s Namespaces] Kubernetes. Namespaces. Kubernetes Official Documentation. URL: <https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/>

[K8s RBAC] Kubernetes. Using RBAC Authorization. Kubernetes Official Documentation. URL: <https://kubernetes.io/docs/reference/access-authn-authz/rbac/>

[K8s Service Accounts] Kubernetes. Service Accounts. Kubernetes Official Documentation. URL: <https://kubernetes.io/docs/concepts/security/service-accounts/>

[kube-scheduler] Kubernetes. kube-scheduler. Kubernetes Official Documentation. URL: <https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/>

[Kubernetes] Kubernetes. Official Documentation. URL: <https://kubernetes.io/docs/home/>

[Kubespary] Kubernetes SIGs. Kubespary: Deploy a Production-Ready Kubernetes Cluster. URL: <https://kubespary.io/#/>

[Lim et al., 2020] Lim, W. Y. B., et al. (2020). Federated Learning in Mobile Edge Networks: A Comprehensive Survey. IEEE Communications Surveys & Tutorials, vol. 22, no. 3, pp. 2031–2063. DOI: 10.1109/COMST.2020.2986024

[Linux namespaces] Linux Programmer's Manual. namespaces(7) - overview of Linux namespaces. The Linux Kernel Archives. URL: <https://man7.org/linux/man-pages/man7/namespaces.7.html>

[Locust] Locust. An open source load testing tool. URL: <https://locust.io/>

[Locustfile] OpenTelemetry. locustfile.py (Load Generator). URL: <https://github.com/open-telemetry/opentelemetry-demo/blob/main/src/load-generator/locustfile.py>

[MinIO] MinIO. High-Performance Object Storage compatible with the Amazon S3 API. URL: <https://min.io/docs/minio/kubernetes/upstream/index.html>

[Minna & Massacci, 2023] Minna, F., & Massacci, F. (2023). SoK: Run-time security for cloud microservices. Are we there yet? Computers & Security, vol. 127, 103119. DOI: 10.1016/j.cose.2023.103119

[Nedelkoski et al., 2019] Nedelkoski, S., Bogatinovski, J., Acker, A., Kao, O., & Dustdar, S. (2019). Anomaly Detection and Classification using Distributed Tracing and Deep Learning. In 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID) (pp. 241–250). DOI: 10.1109/CCGRID.2019.00038

[OpenTelemetry] OpenTelemetry. High-quality, ubiquitous, and portable telemetry to enable effective observability. URL: <https://opentelemetry.io/docs/>

[OpenTelemetry API] OpenTelemetry. Tracing API - Traces and Spans Concepts. URL: <https://opentelemetry.io/docs/concepts/signals/traces/>

[OpenTelemetry Arch] OpenTelemetry. Architecture Guidelines - Collector Deployment Patterns: Gateway vs. Agent. URL: <https://opentelemetry.io/docs/collector/deployment/>

[OpenTelemetry Collector] OpenTelemetry. Collector Architecture & Components. URL: <https://opentelemetry.io/docs/collector/>

[OpenTelemetry Demo] OpenTelemetry. Microservice-based application to demonstrate OpenTelemetry implementation. URL: <https://github.com/open-telemetry/opentelemetry-demo>

[OTEL Helm Chart] OpenTelemetry. OpenTelemetry Demo Helm Chart - values.yaml. URL: <https://github.com/open-telemetry/opentelemetry-helm-charts/blob/main/charts/opentelemetry-demo/values.yaml>

[OTEL Tracing Concepts] OpenTelemetry. Distributed Tracing Concepts. URL: <https://opentelemetry.io/docs/concepts/observability-primer/#distributed-traces>

[Prometheus] Prometheus. Monitoring system & time series database. URL: <https://prometheus.io/>

[Python tracemalloc] Python Software Foundation. tracemalloc — Trace memory allocations. Python 3.12 Documentation. URL: <https://docs.python.org/3/library/tracemalloc.html>

[Satyanarayanan, 2017] Satyanarayanan, M. (2017). The Emergence of Edge Computing. IEEE Computer.

[Shi et al., 2016] Shi, W., Cao, J., Zhang, Q., Li, Y., & Xu, L. (2016). Edge Computing: Vision and Challenges. IEEE Internet of Things Journal. DOI: 10.1109/JIOT.2016.2579198

[SLICES-RI] SLICES-RI. Large-scale scientific infrastructure for experimental studies in computer science and communications.

[SLICES-RI MinIO] SLICES-RI. WEB MINIO. URL: <https://s3-gent.slices-be.eu/minio/ui/browser/>

[W3C Trace Context] W3C. Trace Context - W3C Recommendation. URL: <https://www.w3.org/TR/trace-context/>

[Xing et al., 2025] Xing, S., Wang, Y., & Liu, W. (2025). Multi-Dimensional Anomaly Detection and Fault Localization in Microservice Architectures: A Dual-Channel Deep Learning Approach

with Causal Inference for Intelligent Sensing. URL: <https://pmc.ncbi.nlm.nih.gov/articles/PMC12157131/>

[Zhou et al., 2021] Zhou, X., Peng, X., Xie, T., et al. (2021). Fault Analysis and Debugging of Microservice Systems: Industrial Survey, Benchmark System, and Empirical Study. IEEE Transactions on Software Engineering, vol. 47, no. 2, pp. 243–260. DOI: 10.1109/TSE.2018.2885422

4.2 UGA

4.2.1 Introduction

UGA works on the design and implementation of an anomaly detection solution, with a focus (but not limited to) performance anomalies. The solution analyses metrics coming from different sources (application, system, hardware) to detect anomalies.

Beyond the accuracy of detection, we want to provide a solution that is easy to deploy and use in production. This implies that the models should be easy to configure. Also to adapt to frequent changes in the platform, they should be fast to train and re-train.

Another major concern that we want to address is the carbon footprint of the anomaly detection solution. Anomaly detection involves several activities that consume hardware resources and energy (collecting and storing monitoring data, training models, inference for anomaly detection). For instance, in a cloud-edge environment, if anomaly detection required a lot of data movements between edge nodes and data centers, this could be problematic. Our goal is to propose solutions to reduce this cost.

4.2.2 Challenges

We want to design a solution that can detect anomalies by analyzing metrics from different sources. This is a problem known as multi-variate time series analysis. The goal is to identify unexpected variations between time series that could be evidence that the system is behaving abnormally. Complex machine learning solutions can be used to detect such unexpected variations. However, these models require large amounts of training data and take a lot of time to train. To design a solution that can be suited for cloud-edge environments, we seek a solution that is flexible. By flexible, we mean that the solution should be able to work at scale. It should be able to adapt quickly when the infrastructure and/or the applications are updated. It should also be easy to configure and it should not consume too much computing resources. The challenge is then to provide a high detection accuracy while remaining flexible.

To achieve this flexibility goal, we identified challenges that need to be addressed. The first challenge is related to the constant evolution of edge-to-cloud platforms, and of the applications they run. Microservices applications can be updated as frequently as several times per day. In this context, anomaly detection solutions need to adapt quickly. For ML-based approaches, it implies that it should be possible to train and re-train them quickly, with little

amount of training data. It also implies that the solution should be able to leverage unlabeled data.

Related to this first challenge, another question is about the data used as input for an anomaly detector. Updates in the platform or in the applications can have an impact on the behavior of the metrics that are collected (hence, the need to retrain models). To better detect anomalies, it is tempting to provide as input to an anomaly detector several metrics coming from different sources, that could allow it to have a global view of the behavior of the system. However, if a model uses many metrics as inputs, it increases the risk that one of its inputs will change its behavior after an update, requiring model retraining. Furthermore, having a large model can be more costly to train. Hence, having small but yet accurate models can be a more promising approach.

The second challenge is related to the configuration of the models. Anomaly detection solutions usually require fine parameter configuration to work accurately. One of these core parameters is the definition of the limit that differentiates a normal from an abnormal behavior. This is even more challenging in a dynamic cloud-edge continuum where the system is very dynamic by nature, and many normal variations will occur during the day. Such variations can be, for instance, simply related to the activity of the users. Hence, if the anomaly detection solution requires system operators to fine-tune parameters to work well, there is a high chance that it will never work well in practice. This is even worth if parameter tuning needs to be done again after changes in the platform. We could even imagine that the selection of the parameters could depend on the characteristics of the edge nodes that are considered.

All the challenges listed above lead to the conclusion that designing a perfect anomaly detector is not possible. A perfect failure detector would be a detector that identifies all anomalies (perfect recall) without raising any false positive alarms (perfect precision). Through our work, we would also like to understand which property is more important in practice (recall or precision) for a detector to be useful in practice.

Beyond the flexibility and the accuracy of the solution, we want to address the issue of the carbon footprint and energy consumption of anomaly detection solutions. To address this issue, we need to consider the hardware and energy consumption related to the collection, the transfer and the storage of data. This point is going to be addressed in a second part of the project.

4.2.3 Work Carried out

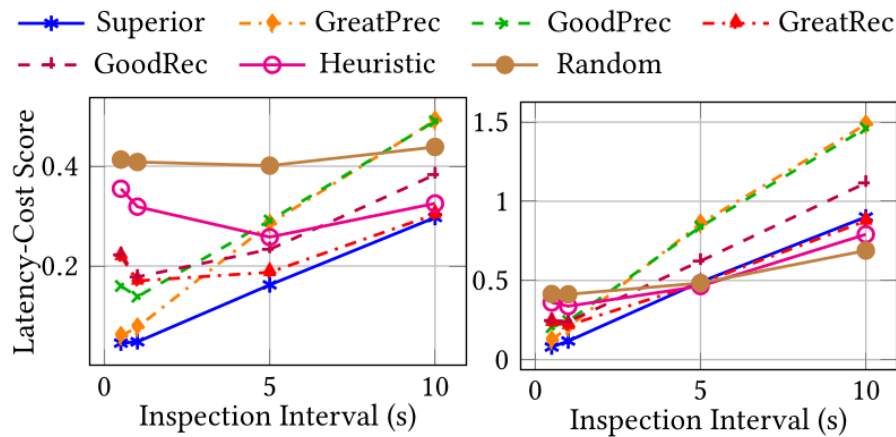
During the first part of the project, our work has focused on the design of a flexible anomaly detection solution based on monitoring metrics timeseries. To this end, we made 2 complementary contributions.

First, we have conducted a study analyzing the characteristics of performance anomaly detectors that are the more useful in practice when operating online cloud services. This study has lead to a publication in the international conference UCC'25 (UCC '25: Proceedings of

the 18th IEEE/ACM International Conference on Utility and Cloud Computing). The full article is accessible here: <https://dl.acm.org/doi/full/10.1145/3773274.3774269>

In this article, we study how recall and precision of an anomaly detector impact the performance (measured as the response time) and the resources consumption of a monitored cloud service. We built an analytical model of such a system, and studied it under different configurations and constraints. Among other things, we considered different frequencies for the anomalies, different service-level objectives (SLO) for the deployed services, etc.

Our main result is presented in the figure below. The Figure presents the performance of a cloud service (computed as a weighted trade-off between achieved latency and amount of resources used) as a function of how frequently the anomaly detection system is able to run. A lower score is better. The results are presented for different detectors (The names of the detectors are self-explanatory. We refer the reader to the full publication for more details). The left graph presents the results when the SLO states that the maximum acceptable latency is 500ms. The other graph considers a tighter SLO (200ms).



(a) Latency SLO of $l_{max} = 500ms$. (b) Latency SLO of $l_{max} = 200ms$.

FIGURE 30 : IMPACT OF THE CHARACTERISTICS OF ANOMALY DETECTORS ON THE PERFORMANCE OF A CLOUD SERVICE

The results show the same trends on both graphs. At low inspection intervals (which is what we target in our work), having a high precision is more important than achieving a high recall. As we can observe on the figures, the "GreatPrec" detector (very good precision but average recall) is almost as good as the "Superior" detector (that has both a very good precision and recall). Interestingly, if a detector is not able to run frequently, recall becomes the most important property.

Beyond this study, we have designed and implemented a new per-service anomaly detection solution. This solution, called Ctl-SRNN, is based on dynamic variational auto-encoders (DVAEs). More specifically, it is based on the Stochastic Recurrent Neural network model, that

we modified to include uncertainty quantification and a control variable that provides context to the model. These modifications have been made to make the solution flexible.

The proposed model detects anomalies in monitoring metrics time series. The control variable provides additional trusted information to the model (such as the current workload) that helps it making the difference between normal and abnormal variations in the metrics. Uncertainty quantification is used to let the model automatically define the bounds of normal metric variations. Based on the estimated uncertainty, the model can define at each time step, the metrics variations that can be considered as normal. The choice of using an SRNN model as basis for our work, is based on the ability of the kind of models to work with timeseries data. More generally, we decided to use a DVAE because of the high generalization capability of such models.

The way Ctl-SRNN works is summarized in the figure below. The SRNN model takes as input the last 30 seconds of monitoring metrics (such as the CPU usage, or the number of packets transmitted by a service) and tries to predict the value of these metrics for the next second, together with uncertainty bounds, that defines the range of acceptable values for the metrics. To make better predictions, the model also receives as input a metric representing the load of the system (here the number of packets received by the service). This metric is used as a control variable by the model, meaning that the model takes the values of this metric as 100% trustable contrary to the other metrics that it tries to predict. An anomaly is detected by the model when the real data get outside of the prediction bounds.

The presented solution is easy to deploy as it only relies for now on metrics that can be easily collected using standard Kubernetes monitoring tools such as cAdvisor and Prometheus.

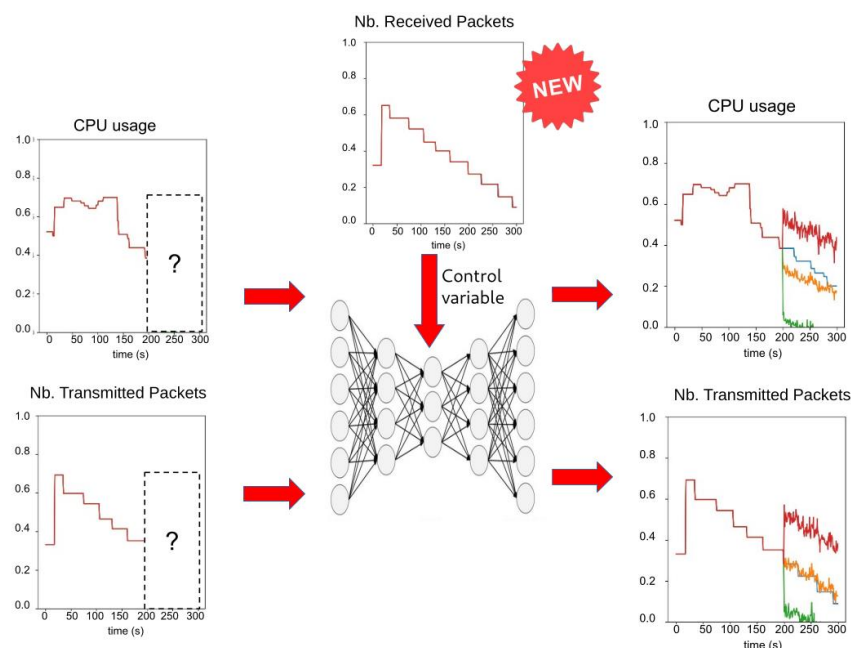


FIGURE 31 : CTL-SRNN: HOW IT WORKS

The results obtained with our model are promising. We are able to train models very fast and with a small amount of data. The models are able to detect performance anomalies correctly without having to be configured. Hence, they are flexible and easy to use in production. Furthermore, they are able to run anomaly detection at the granularity of 1s.

In the coming months, our work will be organized as follows:

- 1) We are currently working on integrating Ctl-SRNN in the Kubernetes eco-system. The next step will be to integrate it with the components designed and implemented by other partners in the E2CC platform.
- 2) We are also working on a detailed evaluation of our solution and its comparison with state-of-the-art anomaly detection solutions
- 3) After these 2 first steps have been addressed, we will focus on the second objective of our work: reducing the carbon footprint of monitoring solutions.

5 Conclusion and Next Steps

The work carried out as part of this deliverable has laid the foundations for an advanced monitoring system tailored to Cloud-Edge environments, by combining complementary approaches related to observability, anomaly detection, and runtime security. The contributions presented demonstrate the implementation of a complete pipeline for collecting, processing, and analyzing traces and metrics, as well as the value of controlled experiments to characterize the behavior of distributed architectures when facing application or infrastructure degradations.

The work conducted by CNRS-LAAS notably made it possible to deploy an experimental Kubernetes platform on the SLICES-RI infrastructure, integrate the OpenTelemetry demonstration application into it, and then implement an observability pipeline based on OpenTelemetry Collector, Jaeger, Prometheus, Grafana, and MinIO. This work also validated the export of telemetry data to persistent Data Lake-type storage, enabled the development of scripts for trace extraction and analysis, and used Chaos Mesh to generate reproducible anomalies. These experiments highlighted the limitations of traditional monitoring tools, particularly when dealing with silent failures, hardware saturation, and propagation effects in microservice architectures.

The work carried out by UGA focused on designing a flexible anomaly detection solution based on metric time series. It led to a study on the impact of precision and recall in anomaly detectors for operating Cloud services, as well as to the design and implementation of the Ctl-SRNN model. This approach, based on dynamic variational autoencoders, aims to provide fast detection that is inexpensive to train, easy to configure, and suitable for production constraints. The initial results show the model's ability to detect performance anomalies with limited training data and without fine-grained manual threshold tuning.

The next steps will consist in consolidating the integration of the developed components within the E2CC platform, in particular the integration of Ctl-SRNN into the Kubernetes ecosystem and its connection with the observability components already implemented. Further effort will also be required to comparatively evaluate the proposed approaches against the state of the art, to build representative datasets from the collected traces and metrics, and to automate the correlation between application events and infrastructure events.

In the longer term, the work should enable progress toward more intelligent and distributed detection mechanisms, capable of taking into account the specific constraints of Edge environments: limited resources, reduced data transfers, controlled energy consumption, and privacy preservation. The identified directions include the use of heterogeneous graphs, Graph Neural Networks for the joint analysis of traces and metrics, and the study of federated learning approaches to bring analysis closer to the data source. These directions will help strengthen the resilience, security, and energy efficiency of the Cloud-Edge infrastructures targeted by the project.

6 Acronyms and Abbreviations

Term	Definition
API	Application Programming Interface
E2CC	Eco Edge to Cloud Computing
RBAC	Role-Based Access Control
VPN	Virtual Private Network

TABLE 1 : ACRONYMS AND ABBREVIATIONS