The Kaiju project: enabling event-driven observability

Conference Paper · July 2020

DOI: 10.1145/3401025.3401740

The Kaiju Project: Enabling Event-Driven Observability

Mario Scrocca
Politecnico di Milano, Italy
mario.scrocca@mail.polimi.it

Riccardo Tommasini
University of Tartu, Estonia
riccardo.tommasini@ut.ee

Alessandro Margara
Politecnico di Milano, Italy
alessandro.margara@polimi.it

Emanuele Della Valle
Politecnico di Milano, Italy
emanuele.dellavalle@polimi.it

Sherif Sakr
University of Tartu, Estonia
sherif.sakr@ut.ee

ABSTRACT

Microservices architectures are getting momentum. Even small and medium-size companies are migrating towards cloud-based distributed solutions supported by lightweight virtualization techniques, containers, and orchestration systems. In this context, understanding the system behavior at runtime is critical to promptly react to errors. Unfortunately, traditional monitoring techniques are not adequate for such complex and dynamic environments. Therefore, a new challenge, namely observability, emerged from precise industrial needs: expose and make sense of the system behavior at runtime. In this paper, we investigate observability as a research problem. We discuss the benefits of events as a unified abstraction for metrics, logs, and trace data, and the advantages of employing event stream processing techniques and tools in this context. We show that an event-based approach enables understanding the system behavior in near real-time more effectively than state-of-the-art solutions in the field. We implement our model in the Kaiju system and we validate it against a realistic deployment supported by a software company.

CCS CONCEPTS

• **Information systems** → *Data streaming*; *Data analytics*; • **Applied computing** → *Event-driven architectures*; • **Computer systems organization** → *Distributed architectures*.

KEYWORDS

Observability, Orchestration systems, Event-based systems, Event stream processing

ACM Reference Format:
Mario Scrocca, Riccardo Tommasini, Alessandro Margara, Emanuele Della Valle, and Sherif Sakr. 2020. The Kaiju Project: Enabling Event-Driven Observability. In *The 14th ACM International Conference on Distributed and Event-based Systems (DEBS '20), July 13–17, 2020, Virtual Event, QC, Canada.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3401025.3401740

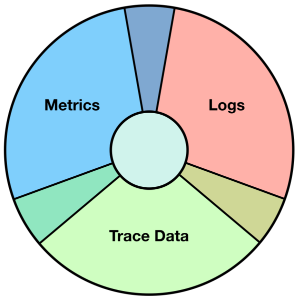

Figure 1: Metrics, logs, and traces are complementary descriptions of the behavior of a software system.

1 INTRODUCTION

With the advent of microservices architectures, the software stacks of modern companies are rapidly increasing in size and complexity, supported by lightweight containerization techniques and orchestration systems such as Kubernetes that make the deployment of distributed systems accessible to even small companies. The complexity of microservices-based architectures calls for modern infrastructure management solutions. Orchestration systems like Kubernetes leverage declarative APIs to simplify the interaction with the infrastructure. However, they limit the developers view on what is happening and what they can control. In this scenario, traditional monitoring techniques, which are often based on a set of predefined trigger rules or dashboards, are insufficient. Thus, developers struggle to deal with a number of problematic scenarios that can emerge at runtime.

The term *observability* was introduced to refer to the problem of *interpreting* the behavior of the overall system [24]. This problem is widespread and not only a prerogative of big companies. Although the term lacks a formalization that supports a systematic investigation, it is based on a precise problem statement: *provide a comprehensive picture of the system behavior integrating its outputs.* Existing observability tools identify three sources of information to collect and process: (i) *metrics*, which are values describing the status of processes and resources of a system, such as per-process memory consumption; (ii) *logs*, which are reports of software execution, and (iii) *traces*, which are composed of causally-related data representing flow of requests in the system. Existing tools struggle to obtain a comprehensive view of the system, since they process metrics, logs, or traces separately. Moreover, they prefer post-hoc to real-time data analysis.

In this paper, we argue that it is of paramount importance to promptly react to failures and continuously adapt the system to

DEBS '20, July 13–17, 2020, Virtual Event, QC, Canada

ACM ISBN 978-1-4503-8028-7/20/07…$15.00
https://doi.org/10.1145/3401025.3401740

mutating environmental conditions. Thus, too little work was done to meet the velocity requirements related to observability. To enable fast reactions to problems, we propose an event-driven conjunct analysis of metrics, logs, and trace data. We show that event-driven observability is an effective approach to understand the behavior of a distributed systems in near real-time. Intuitively, metrics, logs, and traces are events, i.e., notifications of known facts [14], that provide a complementary description of the system behavior (see Figure 1). In summary, we provide the following contributions to the research on observability:

- We coherently survey the state-of-the-art on monitoring, log analysis, and tracing (Section 2).
- We formulate challenges and requirements for the observability problem, with special emphasis on container orchestration systems (Section 3).
- We propose an event-based data and processing model to integrate metrics, logs, and traces, to unify their representation, and to enable near real-time analysis through declarative event stream processing abstractions (Section 4).
- We present Kaiju, an event stream processing engine that we designed in collaboration with SighUp. Kaiju enables near real-time event-based observability in a production-grade container orchestration system (Section 5).
- We put Kaiju into practice (Section 6) and evaluate its overhead on the infrastructure (Section 7).

2 PRELIMINARIES

This section presents the established approaches for (i) monitoring [17], (ii) log inspection [11, 18], and (iii) distributed tracing [9, 20]. In particular, we focus on approaches related to Kubernetes, which is the de-facto standard orchestrator system for containerized applications maintained by the Cloud-Native Computing Foundation (CNCF)[1]. Kubernetes provides a declarative approach based on APIs and manifests. The developer declares the desired deployment state using YAML files, and the orchestration system applies all the necessary changes to reach (and then maintain) the required state starting from the current one.

Kubernetes manages clusters of machines called nodes. Each cluster has at least one *master* node and many *worker* nodes (cf Figure 2). The master node guarantees the status of the deployment in case of failures, updates, and other unforeseeable events. As depicted in Figure 2, the master node runs the core components for the cluster management including the *kube-apiserver*, which exposes the Kubernetes API for accessing the cluster; *etcd*, which is a key-value store containing all cluster data; the *kube-scheduler*, which manages scheduling decisions, and the *kube-controller-manager*, which manages controller processes that implement cluster management functionalities. *Worker* nodes are responsible for running applications, which should be containerized, for instance using Docker[2]. As Figure 2 shows, applications within worker nodes are grouped into *pod*s. Worker nodes communicate with the master node through a *kubelet*, an agent that registers the node to the master's API server. Additionally, they host a *kube-proxy* to handle networking.

Kubernetes optimizes the usage of available resources, provides discovery and load balancing among services, and reacts to events like failures or workload modifications. To these extents, Kubernetes consumes applications logs and metrics about the running applications as well as end-to-end request traces, that span the whole infrastructure. Figure 2 shows how this information is exposed by either the master or the worker nodes.

The different nature of the metrics, logs, and traces led to the development of different tools that optimize for a particular data type. When surveying existing solutions, the following aspects are relevant [12]:

- the *data models and formats* used to represent and encode metrics, logs, and traces, i.e., numeric for metrics, semi-structured text-based for logs, and structured JSON-based for traces;
- the *data collection* mechanisms, which is periodic for metrics, continuous for logs, and reactive for traces;
- the *management* solution, which includes storage, filtering, cleaning, indexing and in general any pre-processing and preparation step that precedes the concrete analysis of the collected data,
- the *data analysis*, which includes data exploration, alerting, querying, or visualization.

In the following, we survey of the state-of-the-art on metrics, logs, and trace data, highlighting the aspects mentioned above.

2.1 Metrics

Metrics are numeric values about the performance of a software infrastructure or application. They are collected using lightweight agent-based mechanisms deployed aside system components or processes in each node [17]. Agents collect data in two ways: (1) directly from running processes, which expose them using ad-hoc libraries, or (2) from the system that is running the process. In the former case, metrics can be customized by the application developer, while in the latter case they mainly relate to resources usage (typically CPU and memory) and networking.

Agents make data accessible to external components responsible for storage and/or processing either in (i) *pull-based* mode, where the receiver asks the agent to forward data, or in a (ii) *push-based* mode, where the agent sends data to the receiver as soon as available. While a push-based approach can guarantee lower latency than a pull-based one, it requires the receiver to cope with incoming data.

In an orchestrated system, metrics are sampled at regular intervals, and typically classified as *custom* metrics, related to the hosted application, and *core* metrics, related to the orchestrator. Relevant core metrics include, but are not limited to: (i) *Node metrics*: metrics related to the hardware resources of the single node (usually exposed by means of tools like *node-exporter*) and metrics related to the *kubelet*. (ii) *Container metrics*: metrics of orchestrated containers. Each node runs a container runtime and, for each node, the kubelet exposes metrics of containers deployed through the *cAdvisor* integrated component. (iii) *Master-node metrics*: metrics related to Kubernetes master components (API Server, Controller Manager, Scheduler, ETCD). (iv) *Network metrics*: metrics from the DNS solution transparently used from Kubernetes (*CoreDNS*), from the *kube-proxy* components deployed in each node, from the cluster network plugin (offering networking between nodes), and from

[1] https://www.cncf.io

[2] https://www.docker.com/resources/what-container

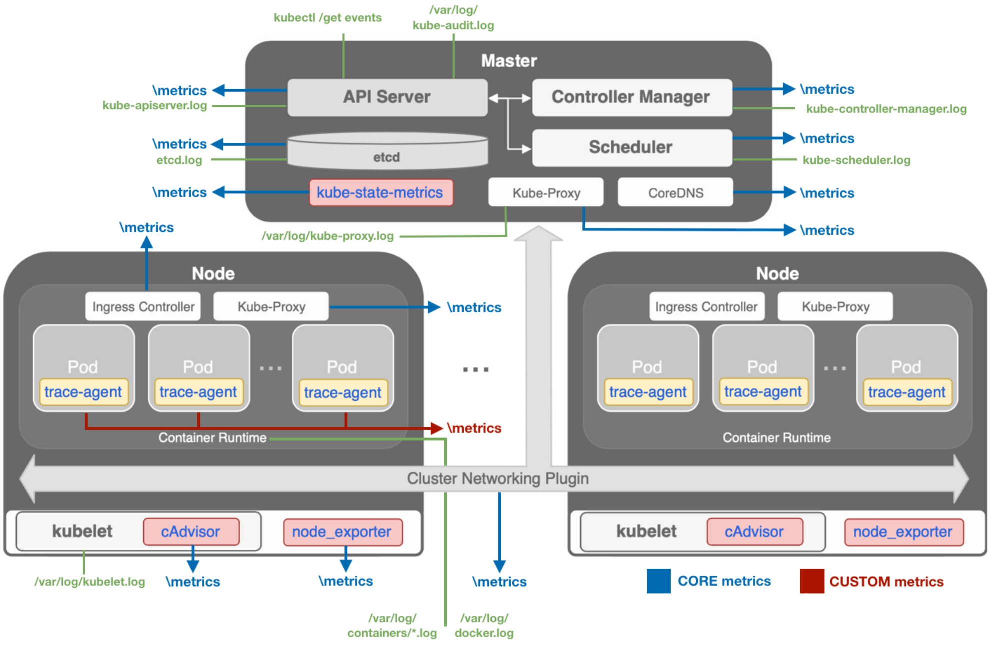

Figure 2: Kubernetes architecture: main components and produced observations.

the ingress controller (if deployed). (v) *Deployment metrics*: data on the state of objects in the cluster (desired and actual state) exposed through the Kubernetes API and made available as metrics from the *kube-state-metrics* addon.

```
api_http_requests_total {
    method="POST",
    handler="/messages"}
    1542650713
    34

api_http_requests_total {
    method="POST",
    handler="/messages"}
    1542651425
    45
```

Listing 1: Two samples of the same metric in Prometheus format.

In Kubernetes, components, such as containers, pods, and deployments, expose by default metrics via HTTP endpoints, which provide a pull-based access mode. Monitoring systems like Prometheus[3] pull metrics endpoints at regular intervals, and store metrics in time-series databases. Kubernetes adopted the Prometheus format (cf Listing 1) as standard for its internal components, while hosted applications need to be manually instrumented to do the same. The Prometheus format for metrics is composed of two main parts: (i) the *id* part identifies the metric through a *name* and a set of key-value pairs (*labels*) to provide additional metadata, and (ii) the *sample* part specifies the timestamped value of the metric.

Metrics data is managed using time-series databases (TSDBs) such as InfluxDB[4]. TSDBs optimize the storage and data access to make data analysis more efficient. The main issue of TSDBs is related to the indexing of data. Metrics are useful at high granularity when they are processed online, for instance to observe spikes in CPU trends. Since in large systems the volume of metrics data is huge, TSDBs must handle high cardinality indexing to support a large number of time series[5]. However, since the relevance of metrics decreases over time, they can be aggregated and stored at lower-granularity for historical analysis. For example, systems like Kapacitor[6] and Gemini2 [2] can perform down-sampling before storing metrics.

The analysis of metric is usually referred to in the literature as *monitoring*. Online monitoring is relevant for use-cases like anomaly detection [21], and it usually employs tools for rule-based alerting. Alerting rules can be *static* or *dynamic*, that is, changing over time based on historical data, on the current values, and on some system parameters. Prometheus allows expressing rules using the PromQL language and their triggering is managed by the *AlertManager* component. Data visualization plays a critical role in the analysis of metrics. Dashboards are the tools to manage alerting rules, query historical data, and observe the system in a meaningful way. State-of-the-art tools widely adopted in the industry include Grafana[7], Kibana for ElasticSearch[8], and Chronograf for the Tick stack[9].

[3] https://prometheus.io/

[4] https://www.influxdata.com/time-series-platform/

[5] https://www.influxdata.com/blog/path-1-billion-time-series-influxdb-high-cardinality-indexing-ready-testing/

[6] https://www.influxdata.com/time-series-platform/kapacitor/

[7] https://grafana.com

[8] https://www.elastic.co/products/kibana

[9] https://www.influxdata.com/time-series-platform/chronograf/

2.2 Logs

Logs are textual records of an operation performed by a software system or an encountered error. They can be structured or unstructured depending on the format of choice. For instance, Listing 2 shows an example of structured log in JSON format.

```
{
    "msg" : "Incoming request",
    "data" : {"clientId":54732},
    "timestamp" : 1541866678,
    "level" : "DEBUG" }
```

Listing 2: Example of structured log in JSON.

In an orchestration system, log are generated at different levels. For instance, Kubernetes provides logs at pod-, node- and cluster-level. Additionally, there are (i) *Logs of containers*: collected from the container runtime if written on standard output. (ii) *Logs of Kubernetes components*: stored on the different nodes. (iii) *Audit logs*: related to requests made to the API Server. (iv) *Kubernetes events*: logs related to events in the cluster (`get events` API).

Like metrics, logs are collected directly from running processes that generate them using ad-hoc libraries and expose them on given endpoints. Differently from metrics, logs are not collected periodically but continuously reported at unpredictable rates. *Fluentd* is the CNCF proposal for log collection. It is a lightweight agent to deploy in each node to build a unified logging layer. Fluentd is configurable through a set of plugins: input plugins specify sources and parsing formats, filter plugins manipulate logs, output plugins specify sinks and output formats. Logs are usually stored before being accessed, and often require pre-processing and filtering. Indeed, log formats may be heterogeneous across the various processes that compose a software system, and this requires parsing strategies, based for example on regex matching, to pre-process them. Moreover, to reduce the amount of data stored, it is often useful to define rules to filter out some logs of a specific service, or with a specific code or level. Efficient solutions to save and retrieve logs, such ElasticSearch[10] exist, but they often introduce a high ingestion delay that limits their capability to sustain high input rates and provide near real-time analysis. To mitigate this problem, message brokers such as Apache Kafka [11] are often used to buffer logs before storing them.

Differently from metrics, a common analysis pipeline for logs is not identifiable. Also a CNCF proposal for log analysis is missing. In a usual deployment, logs are forwarded to data storage and retrieval platforms like ElasticSearch and queried later on using tools like Kibana to provide graphical visualization of the logs gathered, such as the number of log errors. Alspaugh et al. highlight that human inference is crucial to drive log analysis [1] and, thus, it is important to facilitate manual inspection of logs gathered. Recently, tools like Humio[11] have been designed to approach logs as a stream-processing problem providing platforms to ingest and inspect data rapidly and efficiently.

```
{
    "traceID": "f6c3c9fedb846d5", "spanID": "5cfac2ce41efa896", "flags": 1,
    "operationName": "HTTP GET /customer",
    "references": [{"refType": "CHILD_OF", "traceID": "f6c3c9fedb846d5","spanID": "14a3630039c7b19a"}],
    "startTime": 1542877899033598, "duration": 747491,
    "tags": [{"key": "span.kind",
            "type": "string",
            "value": "server"},
            {"key": "http.method",
            "type": "string",
            "value": "GET"}, ...],
    "logs": [{"timestamp": 1542877899033827,
              "fields": [{"key": "event",
                          "type": "string",
                          "value": "HTTP request received"},...]},
             {"timestamp": 1542877899033872,
              "fields": [{"key": "event",
                          "type": "string",
                          "value": "Loading customer"},...]}] }
```

Listing 3: Example of a span collected with the OpenTracing API and serialized by *Jaeger* tracer.

2.3 Traces

Traces are data about requests received by applications. They are produced by specifically instrumented applications that track the request flow. In a Kubernetes infrastructure, the orchestrator does not produce trace data. However, traces are essential to untangle the intricate network of interactions typical of microservices applications. To enable tracing and reconstructing the flow of requests, metadata are usually stored within processes and transmitted in inter-component communications. Recently, distributed tracing tools have been developed for services and microservices architectures with the purpose of retrieving end-to-end data and analyzing the workflow and performance of requests through system components [9, 20]. Open-source tools for trace analysis, such as Jaeger[12] and Zipkin[13], are based on similar pipelines composed by instrumentation libraries in different languages, agents and collectors to gather data, a storage solution, and a visualisation layer.

Two main data models are used to represent data in end-to-end tracing systems: the *span model*, introduced by Google Dapper [20], and the more general *event model*, introduced by X-Trace [7]. By analyzing the two alternatives in details, one can observe that the span model is less expressive than the event model: indeed, each span may be defined as a composition of events, but spans cannot represent every possible composition of events [13]. The CNCF supports the OpenTracing specification[14], a vendor-agnostic effort towards a standardization of instrumentation libraries. The OpenTracing specification defines traces as composed of spans that identify units of computation within the request workflow. A span is composed of metadata to reconstruct the trace (*spanId*, *traceId* and references to other spans), two *timestamps* representing its start time and end time, an *operation name*, a set of *tags* (key-value pairs) and a set of *logs* (key-value pairs with a timestamp) related to it. OpenTracing offers an API for multiple languages, but the serialization format for the data gathered depends on the specific tracer chosen. For this reason the serialization format of the spans

[10] https://www.elastic.co
[11] https://www.humio.com/
[12] https:jaegertracing.io
[13] https://zipkin.io
[14] https://github.com/opentracing/specification

may be slightly different from the OpenTracing specification. Jaeger is the CNCF proposal for trace collection and analysis, compliant with the OpenTracing specification. A Jaeger pipeline consists of: (i) the tracer client library, implemented in the application language, exploiting UDP communication in the same pod to send traces to the *jaeger-agent* deployed as a sidecar component; (ii) the *jaeger-agent* responsible for flushing data to the *jaeger-collector* that writes data to a specific storage solution, such as Cassandra or ElasticSearch; (iii) the *jaeger-query* component interacting with the database to offer an API to retrieve traces and a Gantt visualization of specific traces.

Tracing tools like Jaeger enable a post-hoc analysis of trace data and currently do not implement a solution for online analysis. However, as pointed out in the literature on distributed tracing, online analysis plays an important role [15, 19] and can be effectively paired with the post-hoc one to supervise the system behavior in large-scale systems [9].

3 OBSERVABILITY AS A RESEARCH PROBLEM

The observability problem derives from the industrial need of supervising distributed software systems when the number of components becomes large [16, 22, 24]. To approach observability as a research problem, in the following we identify the challenges, we formulate the research questions, and we present a requirements analysis for the design of methodologies and tools to address this problem. Our inquiry starts from the definition provided by Majors [16], which was also inspired by Kalman[15]:

> *"Observability for software is the property of knowing what is happening inside a distributed application at runtime by simply asking questions from the outside and without the need to modify its code to gain insights."*

Majors refers to the observability problem as the need for gaining visibility of the software behavior from an outside perspective. Thus, we formulate two research problems (OP):

OP1 How can we *expose* the system behavior?
OP2 How can we *make sense* of the system behavior?

In the following, we discuss these two problems and we present a requirement analysis to drive the research on the topic.

3.1 Exposing the System Behavior

The problem of exposing the system behavior (**OP1**) entails defining which data a system must expose. In this context, the trade-off between collecting too much data and not exposing enough information is still little investigated. Therefore, although determining what should be exposed may be dependent on the specific piece of software, we believe that the study of **OP1** should drive the definition of a set of guidelines to investigate the trade-off. To this extent, we need to understand what data constitute the *observable behavior* of the system, that is, the data that can be retrieved as output of the system to infer the system behavior. We can classify the system output along two orthogonal dimensions related to the behavior of the system at runtime, as shown in Figure 3.

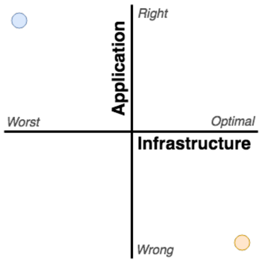

Figure 3: System behavior can be described on the Application and Infrastructure axes.

On the *application* dimension we observe the results of the computation, i.e., application outputs, and we measure their correctness. On the *infrastructure* dimension we observe metrics, logs, and trace data, and we can measure the system efficiency. The two dimensions are orthogonal, yet complementary. Thus, we provide the following definition of observable behavior.

Definition 3.1. The *observable behavior* is the behavior subsumed by all the available output, i.e., application output, metrics, logs, and trace data.

The adjective *observable* highlights that our knowledge of the system behavior is limited to what we can infer from the output. Since we make no assumptions on the knowledge of the application logic, in general, we cannot measure the correctness of the application output. For this reason, **OP1** is mainly related to the infrastructure dimension, i.e., metrics, logs, and trace data, which are agnostic from the application domain.

Different tools exist to deal with metrics, logs, and trace data (see Section 2). However, each tool is often specialized to gather, process, and analyze only one of these types of data. In practice, we lack a unifying conceptual framework and set of tools to obtain an overall perspective on the status of the system. Thus, by treating metrics, logs, and trace data separately we miss an opportunity. We claim that what is missing is a unifying abstraction that fosters interoperability across processing pipelines for metrics, logs, and trace data. To this extent, we elicit from **OP1** the following requirements for a unifying data model for metrics, logs, and trace data. Henceforth we collectively refer to them as *observations.*

R1 A data model must be time-dependent. Observations describe the behavior of the system over time and so their content is strictly bound to specific instants in time: the instant when a metric is collected, the instant when a log is reported, the instant when a span composing a trace has been executed. Accordingly, a suitable data and processing model should include time and temporal relations as first-class concepts.

R2 A data model must be flexible. To empower structural interoperability of observations, the schema should be applicable to the different types of data: metrics content is a numeric value, logs content is a report with different formats under different specifications, trace content can be defined in multiple ways. Moreover, it should be flexible enough to capture the complexity of each domain.

[15] In the context of control theory, Kalman provides a formal definition of observability as a measure of the knowledge about internal states of a system that can be inferred by mean of its external outputs [10].

R3 A data model must be extensible. To empower semantic interoperability of observations gathered from different components, the vocabulary used for describing the environment should be shared. In this way, we can guarantee a common meaning across components, for example operation naming, resource naming, metrics naming.

3.2 Making Sense of the System Behavior

The problem of making sense of the system (observable) behavior (**OP2**) requires to determine what kind of analysis provides the actionable insights. In general, answering **OP2** is about identifying methods to derive insights by *interpreting* the observable behavior of a system. More formally, we define an interpretation as follows.

Definition 3.2. An *interpretation* is a function of the observable behavior of the system over time.

Since distributed systems fail in complex ways [4], no effective test can be done to ensure coverage of all the possible situations and facets system might exhibit. Existing use-cases describe both post-hoc and on-the-fly analyses. An example of the former consists in applying machine learning algorithms on historical data to derive thresholds and models for the system behavior. An example of the latter consists in detecting increasing latency and/or checking Service Level Agreement (SLA) requirements at runtime. As the complexity of the systems we build increases, analyzing software in production environments [16] becomes the only option. Because of this, on-the-fly analysis is gaining popularity, as it provides a prompt reaction to erroneous behaviors. Thus, without neglecting the value of persist data, we claim that making sense of system behavior in near real-time is the preferred solution.

To this extent, we elicit from **OP2** the following requirements for an analysis framework.

R4 The analysis must consider temporal relations. As stated in *R1*, observations depend on time. Therefore, we must consider temporal aspects when we interpret observations.
R5 The analysis must be reactive. The importance of near real-time supervision of systems mainly targets the need to observe and process the current status of the system [2, 21]. We should be reactive and minimize the delay for observation and interpretation.
R6 The analysis must handle different levels of complexity. The analysis framework must handle the variety of data represented using observations. Additionally, it must enable both fine-grained data access and aggregations to effectively transform, filter, correlate, and identify relevant patterns of observations.

4 EVENT-BASED OBSERVABILITY

In this section, we propose a novel solution for the observability problem formulated in Section 3. We name our proposal *event-based observability*, since we advocate for the concept of *event* as the missing data abstraction towards a unified observability pipeline. Additionally, since observations are intrinsically ephemeral data that must be processed on-the-fly [22], we advocate for an event stream processing model as a valuable option to analyse observations on-the-fly before they lose value.

4.1 Data Model

We present the data model for event-based observability, and we show how it fulfills the requirements in Section 3. We propose the concept of *event* defined as follows as a unifying concept for metrics, logs, and trace data.

Definition 4.1. An event is characterized by

`Timestamp` | `Payload` | `Context`

where:

- `Timestamp` is a numeric value representing the event time or the validity interval, e.g., sampling-time for metrics, creation timestamp for logs, event time (events) or completion time (spans) for traces.
- `Payload` is a generic set of key-value pairs that encode the actual information carried by the event (for example, a numeric value observed in some software component).
- `Context` is a generic set of key-value pairs providing additional metadata contextualizing the event (for example, dimensions and scopes of the event like service name, availability zone, instance type, software version, etc.).

Our proposal fulfills the requirement analysis elicited on **OP1**. Indeed, events are time-annotated (*R1*). They adopt a key-value structure which is flexible enough to represent metrics, logs, and trace data (*R2*). Finally, they are extensible through the addition of contextual information (*R3*). Listings 4, 5, and 6 show how the examples from Section 2 look like in the proposed data model.

```
{
  "timestamp": 1542650713000,
  "payload": {
    "name": "api_http_requests_total",
    "value": "34" },
  "context": {
    "method": "POST",
    "handler": "messages" } }
```

Listing 4: Example of metrics as observability events.

```
{ "timestamp": 1541866678000000,
  "payload": {
    "msg": "Incoming request",
    "level": "DEBUG"
  },
  "context": {
    "clientId": 54732
  } }
```

Listing 5: Example of a log as an observability event.

4.2 Processing Model

We propose event stream processing (ESP) as a suitable processing model to enable on-the-fly interpretation of the observable behavior of a software systems by filtering, aggregating, joining, and correlating information from multiple sources [5].

Equation 1 shows the most general definition of interpretation starting from observations

$$f(\mathrm{M}(t),\ \mathrm{L}(t),\ \mathrm{T}(t)) = f(t, \mathrm{M}, \mathrm{L}, \mathrm{T}) \tag{1}$$

```
{ "timestamp": 1542877899033598,
  "payload": {
    "duration": 747491,
    "references": [{
        "refType": "CHILD_OF",
        "traceId": "f6c3c9fedb846d5",
        "spanId": "14a3630039c7b19a"}],
    "operationName": "HTTP GET /customer" },
  "context": {
    "traceId": "f6c3c9fedb846d5",
    "spanId": "5cfac2ce41efa896",
    "operationName": "DEBUG",
    "flags" : 1,
    "span.kind": "server",
    "http.method": "GET",
    "http.url": "/customer?customer=392",
    "component": "net/http",
    "http.status_code": 200 } }
```

Listing 6: Example of a span as an observability event.

where t indicates time, M indicates metrics, L indicates logs, and T indicates traces data.

In general, we refer to ESP as a processing abstraction to continuously provide updated results to a set of standing *queries* (or processing *rules*) [5]. Several types of formalisms exist for queries, which involve different processing mechanisms. Specifically, widespread formalisms accommodate *analytics* on streaming data and *event-pattern recognition*. The former transform input data to update the results of some computation. They deal with the unbounded nature of the stream by relying on incremental computations or by using windowing operators to isolate a recent portion of the stream as input for the computation. The latter look for temporal patterns in the input stream of events [6, 8].

ESP abstractions capture the requirements in Section 3. ESP analysis depend on time (*R4*). Streaming analytics can access temporal attributes and windows are typically defined over the temporal domain. Even more significantly, temporal relations are a core building component for event-pattern recognition. ESP analysis is reactive (*R5*). Computations are reactively triggered by the incoming events, which update output results in the case of analytics queries and initiate pattern detection in the case of pattern recognition rules. ESP abstractions are expressive (*R6*). They enable data, filtering, transformation, joining, correlation, and enrichment with static data. They can capture complex temporal relations among observed events. Remarkably, lots of research efforts were devoted to define simple languages, and today ESP engines such as Esper[16], Drools Fusion[17], and Siddhi [23], provide powerful abstractions for stream analytics and event recognition into high-level declarative languages. Additionally, ESP is well integrated into the Big Data landscape [3, 8, 26].

5 THE KAJIU PROJECT

In this section, we describe the design of the Kaiju project to apply event-driven observability in the concrete context of companies embracing the CNCF stack. The investigation follows the *design science* methodology [25] and was carried on in the industrial context provided by our stakeholders, SighUp[18].

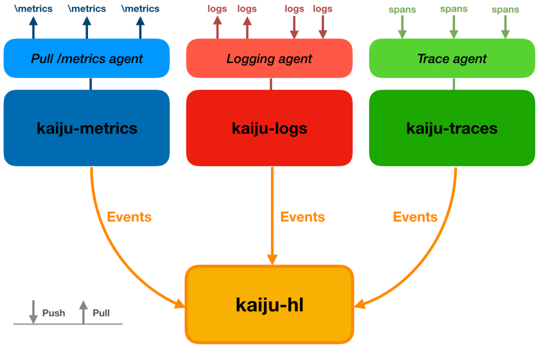

Figure 4: The Kaiju architecture and its main components.

5.1 The CNCF Stack

We implemented the Kaiju project as an extension of the CNCF stack, which includes the following systems: Prometheus for monitoring, Fluentd for logs, and Jaeger for end-to-end traces. We identified the following limitations in the CNCF stack:

- It requires the deployment of three different pipelines to analyze metrics, logs, and traces.
- The pipelines require large amounts of resources, making downsampling almost always necessary.
- The pipelines are not reactive, and require developers to monitor multiple dashboards.
- The integration of data gathered from different tools is almost impossible since metrics, traces, and logs are processed in isolation.
- The signals (events) that the Kubernetes orchestrator system produces at runtime are not collected, preventing their integration with metric, logs, and traces to gain a complete view of the behavior of the system.
- The tools of choice do not provide abstractions and mechanisms to reason on low-level data and extract higher-level information.

5.2 Evolving the Stack with Kaiju

To enable event-driven observability, we extended the CNCF stack to use the data and processing model defined in Section 4.

5.2.1 Requirements. To allow for a simple integration within the CNCF stack and to simplify the migration of existing software systems, our design targeted the following requirements, in addition to the general requirements for observability identified in Section 3.

- Produce an easily-pluggable solution for companies already implementing projects of the CNCF stack.
- Implement a modular solution to accommodate the processing pipelines that deal with metrics, logs, and traces.
- Enable the definition of custom event types to allow high-level processing of data produced by different components.

5.2.2 Architecture. The above requirements led to the design of the Kaiju ESP architecture represented in Figure 4. The architecture consists of multiple modules that can scale independently. The *kaiju-metrics*, *kaiju-logs*, *kaiju-traces* modules consume observations from the sources of metrics, logs, and traces, respectively. They convert these observations into events that adhere to the data model and

[16] http://www.espertech.com/
[17] https://www.drools.org
[18] https://sighup.io/

format presented in Section 4 and perform initial pre-processing when needed.

The *kaiju-hl* module collects events and analyzes them by (i) integrating data from multiple sources, and (ii) abstracting away from specific technicalities of metrics, logs, and traces.

In line with the processing model discussed in Section 4.2, we assume analytical tasks to be expressed using high-level ESP abstractions and carried out by an ESP engine that implements established algorithms and techniques for event stream processing.

Although conceptually represented as a single module in Figure 4, in a concrete deployment *kaiju-hl* can consist of multiple components, each of them focusing on specific analytical tasks, and possibly implemented using different engines. Moreover, a *kaiju-hl* component can produce events for other *kaiju-hl* components, thus enabling a hierarchical decomposition of the analytical problems.

5.2.3 Prototype implementation. In the following, we describe a concrete implementation of the Kaiju architecture defined above[19]. Our prototype implements the *kaiju-metrics*, *kaiju-logs*, *kaiju-traces*, and *kaiju-hl* modules using the Esper open-source ESP engine[20]. Esper is implemented as a lightweight Java library and provides a high-level and expressive Event Processing Language (EPL) to write rules that integrate both stream analytics and event-pattern recognition functionalities.

```
1 create schema A(field1 String, field2 int)
2 create schema B(field1 String, field2 int)
3 create schema D(field3 double)
4 create schema C(field4 string) inherits D
5
6 insert into Output select * from A#time(10 sec)
7 select * from pattern [ a=A -> b=B timer:within(2min)]
```

Listing 7: Primer on EPL syntax.

Listing 7 shows an example of EPL program with few representative queries (statements, in EPL jargon). The EPL statements from line 1 to line 4 define the schema for the streams A, B, C, and D. Schema definition works like in relational databases: a schema consists of uniquely named and strongly typed fields. Schema inheritance is also supported, as exemplified in line 4.

Lines 6 and 7 show two valid EPL queries. Line 6 presents a simple selection query. The stream is identified by its schema name, as tables in relational databases. `#time(10sec)` indicates a sliding window of 10 seconds. The statement shows an `insert into` clause that forwards the query result to a named output stream[21]. Line 7 shows an example of event-pattern recognition. The query looks for an event of type `A` followed-by an event of type `B`; `timer:within(2 min)` indicates the interval of validity of the recognition.

Being implemented as a Java library, EPL is an object-oriented language that supports a POJO presentation of the events. Below, we detail the implementation for each of the module in Figure 4.

kaiju-metrics receives metrics from agents that pull Prometheus endpoints. We adopt Telegraf agents[22] from InfluxData and its plugin to scrape Prometheus endpoints. We model each metric as a POJO taking into account the Telegraf output format (cf Listing 8): the timestamp when metrics are collected, a name, a key-value map of tags as labels and a key-value map of fields representing values sampled. Additionally, *kajiu-metrics* add Kubernetes-related data collected from the cAdvisor internal monitoring component.

```
public class Metric {
  public String name;
  public Long timestamp;
  public Map<String, Float> fields;
  public Map<String, String> tags ; }
```

Listing 8: Metric event POJO.

```
public class Log {
  public Map<String, Object> fields; }
```

Listing 9: Log event POJO.

kaiju-logs receives logs from Fluentd. We implemented a Fluentd output plugin that extracts and forwards data to *kaiju-logs*. The log format depends on the specific library adopted. For this reason, we model each log as a POJO with a very generic key-value map of fields (cf Listing 9). We choose to adopt the ESP engine ingestion time as timestamp to process them. Additionally, *kajiu-logs* exploits the kubernetes-metadata filter plugin to enrich container log records with pod metadata and namespaces.

```
struct Span {
  i64 parentSpanId
  list<SpanRef> references
  i64 startTime
  i64 duration
  list<Tag> tags
  list<Log> logs
  [...] }

struct Process { # Process emitting spans
  required string    serviceName
  optional list<Tag> tags }

struct Batch { # Collection of spans reported by
    processes.
  required Process    process
  required list<Span> spans }
```

Listing 10: Span event POJO: struct specification.

kaiju-traces receives spans from a jaeger-agent deployed as a sidecar process. Jaeger-agents receive spans from processes in-push based manner on a UDP connection. They forward collections of spans called *batches* to collectors. We configured *kaiju-traces* to received spans from the jaeger-agent as a collector. To this extent, we model Jeager data structures as POJOs from their thrift specification in Listing 10. Listing 10 shows the main data structures: `Span`, `Process`, and `Batch`. We consider as timestamp for spans the ESP ingestion time. Additionally, *kajiu-traces* adds metadata about pod, node, and namespace within the modified jeager-agent.

Figure 5 summarizes the flow of events in Kaiju. *kaiju-metrics*, *kaiju-logs*, and *kaiju-traces* directly interact with adapters in the CNCF stack, convert metrics, logs, and traces into the `Metric`, `Log`, and `Span` event types presented above, and perform some initial

[19] Kaiju is available as an open-source project at https://github.com/marioscrock/Kaiju
[20] http://www.espertech.com/esper/
[21] EPL supports schema inference for the output stream
[22] https://www.influxdata.com/time-series-platform/telegraf/

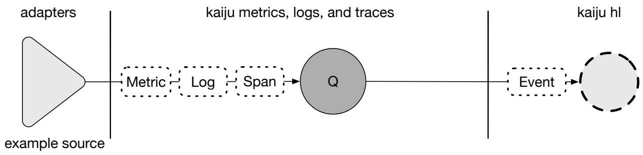

Figure 5: Flow of events in Kaiju

```
public class Event {
  Long timestamp;
  Map<String, Object> payload;
  Map<String, Object> context; }
```

Listing 11: Event POJO.

Alias	Variable
cont_res_limit_cores	kube_pod_container_resource_limits_cpu_cores
cont_cpu_usage_tot	container_cpu_usage_seconds_total
’e.o’	’event.involvedObject’

Table 1: Aliases for variable names.

pre-processing using one or more queries (Q in Figure 5). They convert the results into generic `Events` as defined by the POJO in Listing 11, and forward them to *kaiju-hl* over a TCP connection. *kaiju-hl* performs analytical tasks by integrating data from the other components.

To enable for a better decomposition of the problem, *kaiju-hl* supports the specification of custom `Event` subtypes. It automatically translates custom event specifications into (i) a schema statement determining the event type, and (ii) a processing statement that consumes input `Events`, checks the compliance of the payload and context with the defined event type, and forwards the event on the stream defined by the for that type.

6 PUTTING KAIJU IN PRACTICE AT SIGHUP

To validate the value of Kaiju in practice, we deployed it at SighUp, a software company that adopts microservices architectures and the CNCF stack described in Section 5.1. Our stakeholders provided insights about the kinds of analysis they were interested in. In the following, we present some of the EPL queries that we deployed on Kaiju to satisfy these needs, and we discuss the main lessons we learned.

Metrics. Classical queries on metrics are related mainly to the USE (Utilisation, Saturation, Errors) and RED (Requests, Errors, Duration) methods prescribing the set of values that should be monitored in a software system [22]. Orchestration systems, and Kubernetes in particular, introduce additional metrics that enable queries related to the expected and actual deployment of the system. For instance, they enable checking resources utilization of pods, the number of replicas for a given service, if some pod is in a restart loop, or if the developer can reschedule all pods in case of a node failure.

The EPL language we use in Kaiju can express all the rules above. For instance, Listings 12 and 13 show a first EPL query in *kaiju-metrics* (Q1). Q1 detects those containers that are overcoming the desired CPU usage limit. Listing 12 shows the first part of the rule that constructs the `ContainerCPUUtilization` stream from the stream of metrics. This stream is used internally by *kaiju-metrics* to perform pre-processing. In particular, metrics are retrieved from Prometheus periodically (every second in the SighUp deployment) and maintained as growing counters. Therefore, as shown in Listing 12, query Q1 creates a `ContainerCPUUtilization` event for each subsequent pair of `Metric` events with same tags and related to the usage of cpu in a container (`container_cpu_usage_seconds_total`), where the `usage` is the difference between the values provided by the two readings.

```
create schema ContainerCPUUtilisation(container String,
    pod String, ns String, usage double);

insert into ContainerCPUUtilisation
select m1.tags('container_name') as container,
    m1.tags('pod_name') as pod, m1.tags('namespace') as
    ns, avg((m2.fields('counter') -
    m1.fields('counter'))/((m2.timestamp -
    m1.timestamp)/1000)) as usage
from pattern [every
    m1=Metric(name='cont_cpu_usage__tot') ->
    m2=Metric(m2.name=m1.name and
    m2.tags=m1.tags)]#time(1min)
group by m1.tags('container_name'), m1.tags('pod_name'),
    m1.tags('namespace') output last every 1min
```

Listing 12: (Q1a) ContainerCPUUtilisation.

The EPL statement in Listing 13 constructs the timestamp, payload, and context of the event representing the high CPU usage. Notably, this analysis integrates data from the container runtime in nodes and data from the Kubernetes API (modeled as metrics from the kube-state-metrics plugin). Figure 6 shows the event flow.

Logs. Log analysis is application-dependent, however, classical types of analysis are related to the number of logs in each logging level and to the detection of error messages. Kubernetes introduces two useful log files: the audit log of requests made to the Kubernetes API, and the log of Kubernetes events[23]. The audit log enables detecting unauthorized attempts to access the cluster, while the latter reports all actions done by Kubernetes on containers, pods, images and Kubelets.

Listing 14 shows an example query Q2 processing Kubernetes logs. Q2 retrieves events that correspond to a container image pulled from the registry (`PulledImage` event). Figure 7 shows the event flow for Q2.

Traces. End-to-end traces collected through Jaeger are usually visualized through a Gantt representation to identify bottlenecks and inspect failed requests. However, it is common to query trace

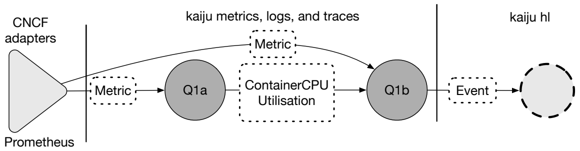

Figure 6: Event-flow for Q1 cf Listings 12 and 13.

[23] Kubernetes events https://github.com/kubernetes/kubernetes/blob/master/pkg/kubelet/events/event.go

```
insert into Event
select now() as timestamp, new {container=container,
      highUsage=usage} as payload, new
      {cpuLimit=fields('gauge'), pod=pod, namespace=ns}
      as context
from ContainerCPUUtilisation#lastevent u,
Metric(name='cont_res_limit_cores')#lastevent m
where m.tags('container')=u.container and
      m.tags('pod')=u.pod and m.tags('namespace')=u.ns
      and usage > fields('gauge')
```

Listing 13: (Q1b) HighCPUUsage.

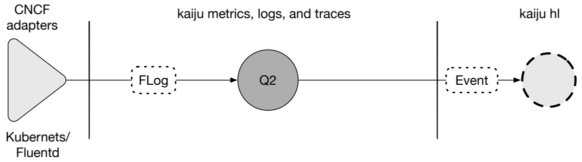

Figure 7: Event-flow for Q2 cf Listing 14.

```
insert into Event
select now() as timestamp,
new { pulledImage=fields('e.o.name')} as payload,
     new{namespace=fields('e.o.namespace')} as context
from FLog(fields('kubernetes.labels.app')='eventrouter'
     and fields('event.reason')='Pulled' and
     (fields('event.message')).contains('pulled'))
```

Listing 14: (Q2) PulledImage.

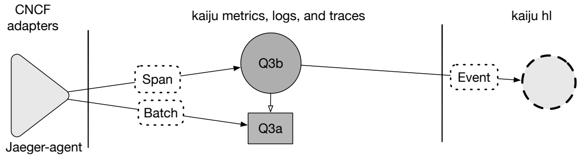

Figure 8: Event-flow for Q3 cf Listings 15 and 16.

data for resource usage attribution and anomaly detection [19]. Listings 15 and 16 show an example of query Q3 that detects spans presenting anomalous latencies. Figure 8 shows the corresponding event flow. Q3a (Listing 15) creates an EPL table `MeanDurationPerOperation` from the stream of `Batches`, which are groups of spans received from the modified jeager agent. The table tracks the mean and variance duration of each operation, and is updated by the `on-merge-update` EPL construct using Welford Online algorithm.

Q3b (Listing 16) uses the table above to identify anomalous spans (`HighLatencySpan`). It implements the so-called *three-sigma rule*, which assumes a Gaussian distribution of samples and detects tails of the distribution fulfilling the equation $(duration - meanDuration) > 3 * stdDev$.

```
create table MeanDurationPerOperation (operationName
      string primary key, meanDuration double, m2 double,
      counter long)

on Span as s
merge MeanDurationPerOperation m
where s.operationName = m.operationName
when matched then update
set counter = (initial.counter + 1), meanDuration =
      (initial.meanDuration + ((s.duration -
      initial.meanDuration)/counter)), m2 = (initial.m2 +
      (s.duration - meanDuration)*(s.duration -
      initial.meanDuration))
when not matched then insert
select s.operationName as operationName,
s.duration as meanDuration, 0 as m2, 1 as counter
```

Listing 15: (Q3a) Consuming Spans.

```
insert into Event
select now() as timestamp,
 //Payload
 new {traceId=traceIdToHex(s.traceIdHigh, s.traceIdLow),
      spanId=Long.toHexString(s.spanId),
      operationName=s.operationName,
      duration=s.duration} as payload,
 //Context
 new {serviceName=p.serviceName, startTime=s.startTime,
      node=s.getTags().firstOf(t ⟹ t.key =
      'kube.node_name').getVStr(),
      pod=s.getTags().firstOf(t ⟹ t.key =
      'kube.pod_name').getVStr(),
      namespace=s.getTags().firstOf(t ⟹ t.key =
      'kube.pod_namespace').getVStr()} as context
from Batch[select process as p, * from spans as s],
MeanDurationPerOperation MDO
where
 (s.duration - MDO[s.operationName].meanDuration) >
 3 * sqrt((MDO[s.operationName].m2) /
      (MDO[s.operationName].counter))
```

Listing 16: (Q3b) HighLatencySpan.

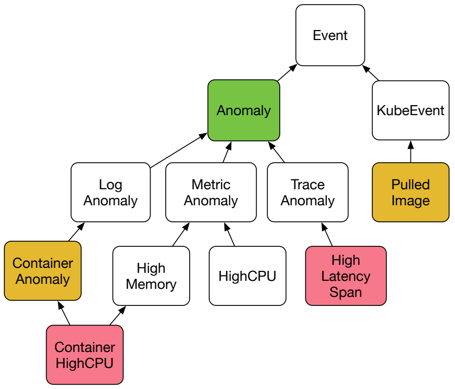

Figure 9: Event hierarchy in Kaiju (partial). Colors indicate events selected in Q4 (green), Q5 (red), and Q6 (yellow).

High-level events. *kaiju-hl* receives `Events` from other components and perform analytical tasks that require integrating data from multiple sources. As discussed in Section 5.2, *kaiju-hl* supports the definition of `Event` subtypes. Figure 9 shows the hierarchy of events we used in our deployment at SighUp. `KubeEvents` identify events related to Kubernetes event-log. `Anomaly` and its subtypes identify critical situations that may happen in the cluster. Listings 17, 18, and 19 show three examples of analysis enabled by *kaiju-hl* using this hierarchy of events. Listing 17 shows a query Q4 that identifies two `Anomalys` (green box in Figure 9) within two minutes.

```
select *
from pattern [Anomaly[2] timer:within(2min)]
```

Listing 17: (Q4) Pattern detecting two anomalies emitted within 2 minutes.

Listing 18 shows a query Q5 that identifies containers with high cpu usage followed by anomalous increase of latencies in traced spans (red boxes in Figure 9).

```
select a.pod as pod
from pattern [a=ContainerHighCPUUsage and
      b=HighLatencySpan(pod = a.pod) where
      timer:within(2min)]
```

Listing 18: (Q5) Pattern detecting high cpu usage and increased latency in spans emitted from the same pod.

Listing 19 shows a query Q6 that combines events and anomalies. Q6 looks for pulled image events followed by container anomalies in the same namespace (yellow boxes in Figure 9). `ContainerHighCPU` event is a subtype of the `ContainerAnomaly` event, but other events may exist and trigger the same pattern (see Figure 9 Query Q6 is useful for example to identify rollouts updating container images and causing wrong system behaviors.

```
select a.pulledImage, b.namespace
from pattern [every a=PulledImageEvent -> every
      b=ContainerAnomaly(namespace=a.namespace) where
      timer:within(10min)]
```

Listing 19: (Q6) Pattern detecting anomalies in containers after the container image has been updated.

In summary, our case study shows that the data and processing models of event-based observability respond to all the requests of our stakeholders using a composition of high-level declarative queries that integrate events coming from different sources and defined at at different levels of abstraction (see again Figure 9).

7 MEASURING KAIJU OVERHEAD

This section reports a quantitative analysis of the performance of our prototype implementation. For our evaluation, we collect all the metric, logs, and traces described in Section 5.1 in the orchestrated system, without performing any down-sampling. We run all our experiments in a cluster composed of four nodes with the following specs: (i) 1 master node with 2 cores x86-64 and 2 GB of RAM; (ii) 3 nodes each with 4 cores x86-64 and 4 GB of RAM.

The basic deployment is based on a micro-service application running in a Kubernetes cluster and comprises:

- Kubernetes to build up a functioning cluster within the SighUp infrastructure;
- components that use the Kubernetes API to emit additional cluster metrics (node-exporter, kube-state-metrics) and logs (eventrouter);
- the Rim app: a modified version of the HotR.O.D.[24] demo app from Jaeger (i) composed of four micro-services deployed separately with each service scaled up to 3 replicas; (ii) enabling configurable introduction of simulated latency / errors through ConfigMap; (iii) enabling load tests; (iv) instrumented with OpenTracing to report traces tagged with Kubernetes metadata;

	Transmitted data
Master node	0.43 Mb/s
Other nodes	1.50 Mb/s

Table 2: Kaiju overhead. Data transmitted per node (avg).

To include the Kaiju artifact we added the following components:

- Kaiju basic deployment with one module for each type;
- the Telegraf and Fluentd agents on each node;
- the custom Jeager agent that collects traces on each pods.

We measure the overhead of Kaiju by comparing the deployment with and without our prototype. In both cases we apply a uniform load test of 30000 requests over 10 minutes to the Rim application (average 50 requests/second). During the tests we measured an average of 2000 metrics/s from *kaiju-metrics*, 1500 logs/s from *kaiju-logs*, 2500 spans/second from *kaiju-traces*.

Figure 10 compares the CPU and memory utilization for each node, without Kaiju (left) and with Kaiju (right), when considering all the rules discussed in Section 6. While both CPU and memory utilization increase, both of them remain well below the maximum resources available in our test deployment, demonstrating the feasibility of event-based observability in practice. Notably, despite Esper retains many events in memory during the analysis the memory utilization of all the nodes increases by only some hundreds MBs. Thus, despite the limited memory resources of our test infrastructure, we could run Kaiju without applying any sampling to the input data.

Table 2 reports Kaiju overhead in terms of network. Again, despite we do not apply any sampling on observations, the additional network bandwidth required to run Kaiju in our test environment is easily manageable.

In summary, the above result makes us confident that on-the-fly event-based observability is possible in production environments without significantly increasing the demand for hardware and network resources with respect to the standard deployment of a microservices-based application.

8 CONCLUSIONS

The number of companies relying on microservices architectures and container orchestration is rapidly increasing. These architectures bring several benefits but demand for new mechanisms and tools to interprete the systems behavior. In this paper, we proposed an event-based approach for observability in the context of container orchestration systems. Our work stems from concrete requirements and demands collected from industrial stakeholders, and led to the development and validation of Kaiju, an Esper-empowered artifact, implementing the proposed data and processing model.

Kaiju provides near real-time processing of data enabling reactivity. It decouples domain knowledge related to metrics, logs, and traces to specific modules but, at the same time, enables to easily integrate and correlate data forwarded from different modules. It offers high-level processing abstractions and enables hierarchical decomposition of the analysis. We evaluated Kaiju both in terms of the benefits it provides to developers by integrating metrics, logs,

[24] https://github.com/jaegertracing/jaeger/tree/v1.5.0/examples/hotrod

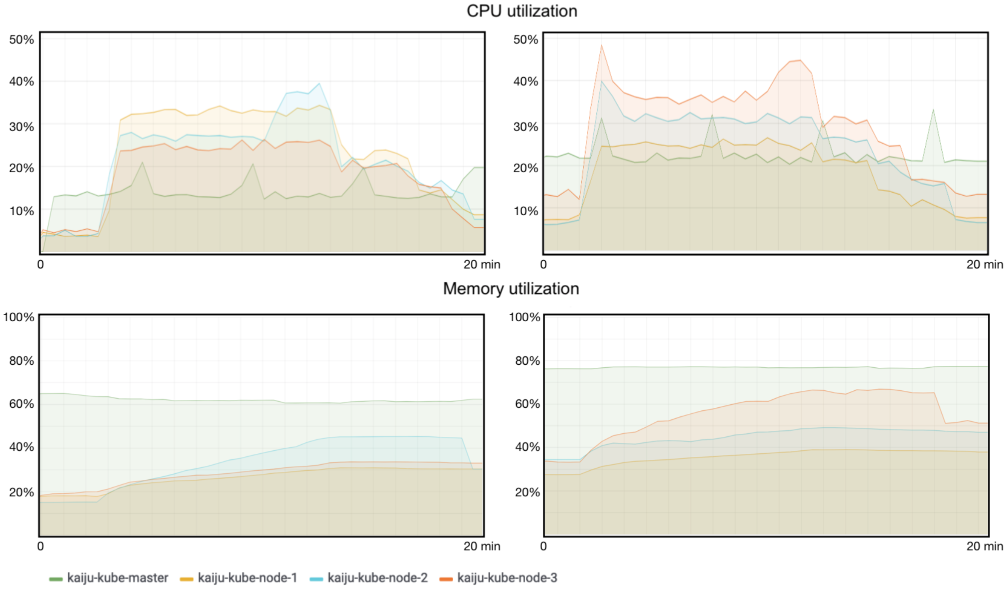

Figure 10: Kaiju overhead. CPU and memory utilization for each node without Kaiju (left) and with Kaiju (right).

and traces, and in terms of overhead. Even considering that Kaiju does not perform any down-sampling, its adoption increasing the load of 20% in the worst case.

In general, we believe that our work has the potential to delineate a new research direction that brings the modeling and processing abstractions of event-based systems into the realm of observability, a key problem for software architects that aim to understand the behavior of increasingly complex and distributed software systems.

ACKNOWLEDGMENTS

Thanks to SighUp and in particular to Jacopo Nardiello.

REFERENCES

[1] Sara Alspaugh, Bei Di Chen, Jessica Lin, Archana Ganapathi, Marti A Hearst, and Randy H Katz. 2014. Analyzing Log Analysis: An Empirical Study of User Log Mining.. In *LISA*. 53–68.

[2] Bartosz Balis, Bartosz Kowalewski, and Marian Bubak. 2011. Real-time Grid monitoring based on complex event processing. *Future Generation Computer Systems* 27, 8 (2011), 1103–1112.

[3] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink: Stream and Batch Processing in a Single Engine. *IEEE Data Engineering Bullettin* 38, 4 (2015), 28–38.

[4] Richard I Cook. 1998. How complex systems fail. *Cognitive Technologies Laboratory, University of Chicago. Chicago IL* (1998).

[5] G. Cugola and A. Margara. 2011. Processing flows of information: From data stream to complex event processing. *Comput. Surveys* 44 (2011), 15:1–15:62.

[6] Opher Etzion and Peter Niblett. 2010. *Event Processing in Action* (1st ed.). Manning Publications Co.

[7] Rodrigo Fonseca, George Porter, Randy H Katz, Scott Shenker, and Ion Stoica. 2007. X-trace: A pervasive network tracing framework. In *Proceedings of the 4th USENIX conference on Networked systems design & implementation*. USENIX Association, 20–20.

[8] Nikos Giatrakos, Elias Alevizos, Alexander Artikis, Antonios Deligiannakis, and Minos Garofalakis. 2019. Complex event recognition in the Big Data era: a survey. *The VLDB Journal* (07 2019).

[9] Jonathan Kaldor, Jonathan Mace, Michał Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, et al. 2017. Canopy: An End-to-End Performance Tracing And Analysis System. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 34–50.

[10] Rudolf Kalman. 1959. On the general theory of control systems. *IRE Transactions on Automatic Control* 4, 3 (1959).

[11] Jay Kreps, Neha Narkhede, Jun Rao, et al. 2011. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*. 1–7.

[12] Łukasz Kufel. 2016. Tools for distributed systems monitoring. *Foundations of Computing and Decision Sciences* 41, 4 (2016), 237–260.

[13] Jonathan Leavitt. 2014. End-to-End Tracing Models: Analysis and Unification.

[14] David Luckham. 2002. *The power of events*. Vol. 204. Addison-Wesley Reading.

[15] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. 2015. Pivot tracing: Dynamic causal monitoring for distributed systems. In *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 378–393.

[16] Charity Majors. 2018. *Observability for Emerging Infra: What Got You Here Won't Get You There*. Youtube. https://www.youtube.com/watch?v=oGC8C9z7TN4 Accessed september 2018.

[17] Matthew L Massie, Brent N Chun, and David E Culler. 2004. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Comput.* 30, 7 (2004), 817–840.

[18] Karthik Nagaraj, Charles Killian, and Jennifer Neville. 2012. Structured comparative analysis of systems logs to diagnose performance problems. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 26–26.

[19] Raja R Sambasivan, Rodrigo Fonseca, Ilari Shafer, and Gregory R Ganger. 2014. So, you want to trace your distributed system? Key design insights from years of practical experience. *Parallel Data Lab., Carnegie Mellon Univ., Pittsburgh, PA, USA, Tech. Rep. CMU-PDL-14* (2014).

[20] Benjamin H Sigelman, Luiz Andre Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. Dapper, a large-scale distributed systems tracing infrastructure. (2010).

[21] Michael Smit, Bradley Simmons, and Marin Litoiu. 2013. Distributed, application-level monitoring for heterogeneous clouds using stream processing. *Future Generation Computer Systems* 29, 8 (2013), 2103–2114.

[22] C. Sridharan. 2018. *Distributed Systems Observability: A Guide to Building Robust Systems*. O'Reilly Media. https://books.google.it/books?id=wwzpuQEACAAJ

[23] Sriskandarajah Suhothayan, Kasun Gajasinghe, Isuru Loku Narangoda, Subash Chaturanga, Srinath Perera, and Vishaka Nanayakkara. 2011. Siddhi: A second look at complex event processing architectures. In *Proceedings of the 2011 ACM workshop on Gateway computing environments*. ACM, 43–50.

[24] Cory Watson. 2018. *Observability at Twitter*. https://blog.twitter.com/engineering/en_us/a/2013/observability-at-twitter.html Accessed September 2018.

[25] Roel J Wieringa. 2014. *Design science methodology for information systems and software engineering*. Springer.

[26] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized Streams: Fault-tolerant Streaming Computation at Scale. In *Proceedings of the Symposium on Operating Systems Principles (SOSP '13)*. ACM, 423–438.