Politecnico Di Milano

Department of Electronic, Informatics and Bioengineering

School of Industrial and Information Engineering

Towards Observability with (RDF) Trace Stream Processing

Supervisor: Prof. Emanuele Della Valle

Co-Supervisor: Riccardo Tommasini

Author: Mario Scrocca

Personal Number: 874830

This dissertation is submitted for the degree of
Master of Science

December 2018

To my family

"Because the people who are crazy enough
to think they can change the world,
are the ones who do."

Abstract

Distributed software systems and cloud-based micro-service solutions are getting momentum. Although this paradigm-shift foster scalability, it is making architectures too complicated to maintain using traditional techniques. How to supervise such complex systems is not apparent, and industry is investigating the subject under the *observability* umbrella. In this thesis, we investigate *observability* as a research problem. First, we provide a definition that clarifies the scientific boundaries. Then we investigate the following research questions: how to expose and how to make sense of the system behaviour at runtime.

In particular, in order to expose the system behaviour: (i) we employ event as a unifying data model for metrics, logs, and trace data, (ii) we explain how to map existing data formats to the proposed model, and (iii) we discuss the benefits of a unified abstraction for observability.

Moreover, in order to make sense of the system behaviour: (i) we elicit a set of requirements, and (ii) we build a proof-of-concept implementing a stream processing solution that satisfies them. Notably, since the state-of-the-art already provides a stream processing solution that makes sense of metrics and logs, we focus on trace stream processing. We followed the *Design Science* framework to realise our proof-of-concept. I.e., we design and implement an artifact (*Kaiju*, a Trace Stream Processor) and we study its interactions in the addressed context (*Rim*, a reproducible environment capable of emulating a distributed system and its typical issues). We provide evidence of the artifact validity comparing it with state-of-the-art distributed tracing tools, and against the typical use cases for such systems recreated in *Rim*.

Last but not least, we discuss the benefits of processing together metrics, logs and trace data. Since stream processing solutions cannot tame the heterogeneity of metrics, logs, and trace data, we employed a stream reasoning approach. In this direction, we propose an ontology to model trace data, and we report an explorative analysis of an RSP engine consuming this type of data.

Estratto

Sistemi software distribuiti e soluzioni a micro-servizi basate sulle tecnologie del cloud presentano una crescente adozione. Sebbene questo cambio di paradigma favorisca la scalabilità, al contempo sta rendendo le architetture troppo complicate per essere gestite attraverso le tecniche tradizionali. Determinare come supervisionare sistemi così complessi non è scontato, e per questo motivo il mondo dell'industria sta approfondendo questa tematica definendola con il nome di *observability*. In questa tesi affrontiamo l'*observability* come un problema di ricerca. Per prima cosa forniamo una definizione che chiarisca il termine da un punto di vista scientifico. Poi affrontiamo le seguenti domande di ricerca: come esporre e come dare significato al comportamento del sistema a runtime.

Per esporre il comportamento del sistema: (i) utilizziamo l'evento come modello dei dati unificante per metriche, logs e tracce, (ii) mostriamo come stabilire una corrispondenza tra i formati dei dati esistenti e il modello proposto, e (iii) discutiamo i benefici di un'astrazione unificante per l'*observability*.

Per dare significato al comportamento del sistema: (i) elicitiamo un insieme di requisiti, e (ii) realizziamo un proof-of-concept che implementi una soluzione di stream processing rispettando i requisiti. Poichè lo stato dell'arte fornisce già una soluzione di stream processing per metriche e logs, ci concentriamo sullo stream processing applicato alle tracce. In questo lavoro ci atteniamo al framework *Design Science* per realizzare il nostro proof-of-concept, i.e., progettiamo ed implementiamo un artefatto (*Kaiju*, un Trace Stream Processor) e studiamo le sue interazioni nel contesto affrontato (*Rim*, un ambiente riproducibile che emuli un sistema distribuito e le sue principali problematiche). Dimostriamo la validità dell'artefatto comparandolo con gli strumenti correntemente utilizzati per il tracing distribuito e in relazione ai casi d'uso tipici di questi sistemi ricreati mediante *Rim*.

In conclusione, consideriamo i benefici nel processare insieme metriche, logs e dati delle tracce. Poichè soluzioni di stream processing non possono contrastare efficacemente l'eterogeneità di metriche, logs e tracce, consideriamo un approccio stream reasoning. In questa direzione, proponiamo un'ontologia per modellare le tracce e riportiamo un'analisi esplorativa di un RSP engine che consumi questo tipo di dato.

Table of contents

List of figures

List of tables

List of listings

Chapter 1

Introduction

In this chapter, we introduce our work on *observability* for software. The need for *observability* is related to the problem of gaining visibility on systems at runtime and has become relevant because of the growing difficulty in maintaining and debugging complex systems architectures. In Section 1.1 we introduce the motivations that led this work, in Section 1.2 we formulate the research questions addressed, and in Section 1.3 we overview the contributions this thesis offers. To conclude, in Section 1.4 we provide the outline of this thesis.

1.1 Motivations

Modern software systems are distributed, and a shift towards cloud-based micro-services architectures is currently ongoing. The development of orchestration systems managing containers and the sustainability of cloud solutions played a central role. In this context, it is not apparent how to supervise running systems made by widely distributed and ephemeral architectures. These systems present a complex network of interactions between components and may exhibit a multitude of possible failure states. Moreover, virtualisation techniques over machines often managed by external providers shifted the focus on the users' experience and the analysis of execution at an application level [48, 58]. Indeed, currently it is more relevant to detect a poor-performed or wrongly-executed user request than being alerted on restarted containers or machines, scenarios often managed automatically by orchestrators. We are still interested in how the software impact on resources available regarding performance but less and less on machine-level monitoring and all layers beside the virtualisation abstraction.

These challenges are still open, and industry aggregates them under the ***observability*** umbrella. Traditional monitoring tools (described in Section 2.1) mainly addresses metrics,

i.e., measurements of resource consumption sampled at regular intervals, and logs, i.e., reports of operations/errors happening in the system. However, these tools fail to provide a complete picture of interactions between services. Thus, the interest in end-to-end distributed tracing tools, focusing on request work-flow, is growing.

Many industrial tools were released aiming at solving the challenge of reactively understanding black-box complex software systems. Nevertheless, a shared definition for the *observability* problem is still missing. Existing approaches were developed facing very specific issues, and they lack the necessary level of abstraction to offer a comprehensive solution.

We recognise a lack of formalisation in describing the *observability* problem and its challenges and, to the best of our knowledge, there aren't work in the literature addressing it specifically. In this thesis, we investigate *observability* as a research problem, we describe the related research questions, we elicit requirements from questions proposed, and we discuss a set of solutions fulfilling them.

1.2 Research questions

Starting from the proposed industrial definitions of *observability*, we consider **observability for distributed software systems** as *the property of a system to expose its behaviour at runtime through its outputs* and we identify two related subproblems. The first subproblem is related to how a running system can *expose* its behaviour, while the second focuses on how to *make sense* of the exposed behaviour.

Traditional monitoring systems focus on one of the three different outputs of the system, i.e., metrics, logs and trace data. We called these outputs *observations*, and we discuss how they provide different perspectives on the problem of exposing the system behaviour. We pose therefore the following research question: *Is it possible to unify the data models and processing pipelines of metrics, logs and trace data (i.e.,* observations*) to provide a single and significative output for the observable behaviour of a software system?*.

Moreover, we discuss the importance of dynamic analysis to supervise the system at runtime, and we pose the following research question: *Is it possible to make sense in near real-time of information needs about the system observable behaviour considering the available* observations *at runtime, and despite data heterogeneity?*.

1.3 Contributions

The two proposed research questions have been addressed in this thesis discussing: (i) a data model for *observations* to *expose* the system behaviour, and (ii) a processing model to *make sense* of system behaviour.

To design a data model for *observations*, we start from an analysis of currently used formats for metrics, logs and trace data highlighting their similarities and differences. We identify the dependency on a timestamp as the unifying aspect and, for this reason, we propose a data model based on the concept of event where time is a first-class citizen. We detail the *observability event* data model showing how the proposed model can map formats currently used for *observations*, and we explain the benefits of a unified perspective.

Researching a processing model for *observations*, we show how a stream processing approach can fulfil requirements identified. The literature on the topic already indicates the effectiveness of a stream-based approach for metrics and logs [8, 40, 58] and many tools exist in the market to deal with those types of data as incoming streams. On the other hand, even if some research works highlights the benefits of dynamic analysis applied to trace data [38, 47], none of them uses a stream processing engine and current state-of-the-art distributed tracing tools focus only on static analysis of data gathered. Therefore, we provide a specification for a Trace Stream Processor (TSP) dealing with trace data as a stream, and, as a proof of concept for our specification, we implement *Kaiju*, a TSP prototype. Following the Design Science framework, proposed in [66], to effectively design an artifact it is necessary to evaluate it in its interaction with the addressed context. To this purpose, not having a real production distributed system nor a dumped dataset, we also provide a specification for *Rim*, a reproducible environment capable of emulating a distributed system and its typical issues.

Once assumed the effectiveness of a stream-based approach dealing separately with metrics, logs or trace data, we investigated the benefits of processing together all *observation* types. However, stream processing solution cannot tame the heterogeneity and, for this reason, we propose the adoption of a *stream reasoning* approach, and in particular of RDF Stream Processing (RSP). RSP engines prescribe how to tame variety and velocity simultaneously [23]. Therefore, an RDF Stream Processing Engine and a model for semantic data integration (ontology) can offer a solution to the problems raised by both research questions. Research works modelling ontologies for logs and working on them as RDF graphs exist in the literature ([26, 52] and RLOG[1]). Moreover, a related work on metrics

[1] RLOG - an RDF Logging Ontology http://persistence.uni-leipzig.org/nlp2rdf/ontologies/rlog/rlog.html#

analysis with stream reasoning exists [50]. Even if it does not define an ontology, given the currently ongoing *OpenMetric* effort towards a standardisation, we leave this as future work. In this work, we exploit the work done with *Kaiju* to design an ontology for trace data based on the *OpenTracing* specification [35]. Moreover, we report an explorative analysis of an RSP engine consuming trace data as RDF graphs.

1.4 Outline of the thesis

This thesis develops as follow:

- **Chapter 2** provides an overview of the main research areas related to this thesis. It describes monitoring and tracing techniques for distributed systems, the *Information Flow Processing* domain and the *Stream Reasoning* research field. It also presents an introduction to the *Design Science* framework.
- **Chapter 3** defines *observability* for software. It identifies the two related research questions addressed in this work, and it elicits the related requirements.
- **Chapter 4** describes design. It discusses a data model and a processing model fulfilling requirements elicited, and it formulates specifications for *Kaiju* and *Rim* prototypes. It also proposes a *stream reasoning* approach to face simultaneously the two research questions proposed.
- **Chapter 5** contains the details of implementation and describes how we realised *Kaiju* and *Rim*. It also describes an ontology for trace data and how we enable forwarding of data gathered from *Kaiju* to an RSP engine.
- **Chapter 6** contains the evaluation made. It describes the deployment used in our experiments, the procedures followed to evaluate *Kaiju* in the *Rim* environment, and a discussion of results. It also describes an explorative analysis of the RSP engine capabilities in processing trace data.
- **Chapter 7** draws the conclusion of this thesis work describing its limitations and the future works.

Chapter 2

Background

In this chapter, we provide an overview of the principal research topics related to this work.

In Section 2.1, we present a summary of currently used methods to monitor distributed systems, reporting tools used in the industry and research works reported in the literature. In particular, to contextualise our design choices in building *Kaiju*, in Section 2.2, we focus on an in-depth review of distributed tracing systems managing data collected from request work-flow. In Section 2.3, we overview the *Information Flow Processing* domain presenting stream-processing alternatives. In Section 2.4, we present the main concepts related to the *stream reasoning* research area. In conclusion, in Section 2.5, we provide also an introduction to the *Design Science* framework followed in designing the two prototypes, *Kaiju* and *Rim*, as an *artifact* and the environment emulating the *context*.

2.1 Monitoring

Systems running software require supervision at runtime to provide performance debugging, failure identification and notification. Existing tools for troubleshooting distributed systems are built exploiting different input data, however, it is possible to point out three main categories: (i) monitoring through **metrics** [49], (ii) **log aggregation/analytics** tools ingesting and analysing logs from different processes/components [51, 40], and (iii) **tracing systems** collecting causally-related data from request work-flow (discussed in details in Section 2.2).

In this section, we will focus on system monitoring through *metrics* and *logs*. Metrics are numeric data on system and application performances sampled at regular intervals. Logs are records, *unstructured* (e.g., plain text) or *structured* (e.g., *JSON*), reporting what happens in the system. The different nature of the data, numeric for metrics and mainly

string-based for logs, has led to the development of different tools trying to optimise the management of the two different types of data. However, all tools have to face the following sets of problems [41]: (i) *collection* from different components, (ii) *storage* to persist data, (iii) *processing* to manipulate data, (iv) *alerting* to enable automatic notification in case of anomalous data, (v) *visualization* to enable inspection of data.

We will discuss these problems concerning metrics and logs and considering both research and industrial works.

Collection A monitoring tool should put in place a collection mechanism to gather data from system components. Highly distributed systems require a scalable approach, as discussed in the design of the *Ganglia* monitoring system [49]. Therefore, a network of daemon *agents*, lightweight processes, are usually installed in each node *to collect* and *forward data* to components collecting them.

Data gathered are usually: (i) exposed at given endpoints by application components (instrumented to this purpose through ad-hoc libraries), or (ii) gathered from the system and mainly related to resources usage (memory, CPU, etc...) and networking. The collector component can be distributed, to avoid networking overhead, or centralised, to avoid fallacies of distributed computing. Moreover, it should be replicable to guarantee scalability.

Data collection can follow two different interaction paradigms, in each step of the monitoring pipeline: (i) *pull*: receiver asks the sender to forward data, (ii) *push*: sender sends data to the receiver as soon as available (guarantees low latency but the receiver should cope with incoming data to avoid to be flooded).

Storage Monitoring data are usually stored before being accessed and processed. Ad-hoc storage solutions have been developed to manage metrics and logs efficiently.

Tools for metrics, like *Prometheus*[1] and the *Tick* stack by *InfluxDB*[2], exploit Time Series Databases (TSDB) to store and retrieve data efficiently. These databases are optimised to handle numeric data and also offer techniques to reduce the storage needed from metrics over time. Granularity reduction operates subsequent aggregations at predefined time intervals and allows to free space. Indeed, metrics are useful at high granularity when they are collected, e.g. to observe spikes in CPU trends, but can be used at lower-granularities for historical analysis since their relevance reduces over time. The main issue TSDB have to face is related to the indexing of data, since in large systems the number of unique

[1] Prometheus https://prometheus.io/docs/
[2] Tick stack https://www.influxdata.com/time-series-platform/

metrics can be in the order of millions, these databases should handle high cardinality indexing to support a large number of time series[3].

On the other hand, logs are higher-volume data and, differently from metrics, they are not collected periodically but reported at variable rates. For this reason, they are often pre-processed and filtered before being stored. Efficient solutions to save and retrieve logs, like *Elasticsearch* [34] exists, but given ingesting delay and high-volumes, they cannot keep high performances for real-time analysis. To solve this issue, when *Kafka* [40] is already integrated into the production environment, it is useful to exploit it approaching logs as a stream of data. In this way it is possible: (i) to enable real-time analytics, (ii) to buffer logs to be stored, and (iii) to directly expire short-term logs avoiding storing them. Moreover, new tools like *Humio*[4] or *Honeycomb*[5] are explicitly designed to approach logs as a stream-processing problem providing platforms to ingest and inspect data rapidly and efficiently.

Processing In most tools, metrics and logs are saved and then queried, visualised or analysed. This type of analysis is called *offline* analysis and processes data statically and a posteriori. However, some use cases, e.g., anomaly detection, requires to process dynamic data *online*, i.e., as soon as they are available [58]. For this reason, as discussed for logs, a stream processing engine can be exploited to pre-process also metrics, offering also the possibility of down-sampling before storing and to perform complex and real-time analytics (e.g., *Kapacitor*[6] or *Gemini2* [8]). Moreover, processing logs and metrics as a stream allows also to query them relying on temporal aspects such as sliding window calculation or event correlation [8].

Alerting In order to be reactive to failures, monitoring tools allows to specify a set of rules to generate alerts. This rules can be executed and checked against the storage regularly or implemented through a stream processor engine pre-processing data. Alerting rules can be *static*, i.e. fixed, or *dynamic*, e.g., varying with respect to historical data or the current value of some system parameters.

In the past, alerts were principally related to *black-box monitoring*, checking system status from an outside perspective to prevent failures of a predictable nature. Currently, instead, some problems as health-checking and load balancing are automatically solved

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

[4] Humio https://www.humio.com/

[5] Honeycomb https://www.honeycomb.io/

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

(thanks to orchestration systems like *Kubernetes*[7]), but, the growing complexity of systems generates a broad set of unpredictable failures. As pointed out by *Alspaugh et al.* analysing *Splunk* usage [3], human inference is a crucial factor to drive the analyses in these cases and therefore, it is important for alerting to provide actionable information facilitating manual inspection of data gathered. The described context asks for *white-box monitoring*, i.e., to express alerting rules and to gather metrics also at the application-level [58].

Visualization The last problem faced by monitoring tools is about the visualisation of data gathered. Often tools provide complete interfaces to deploy alerting rules, query the storage and to make dashboards plotting data in a meaningful way (e.g. open-source *Grafana*[8], *Kibana* for ElasticSearch[9], *Chronograf* for the Tick stack[10]).

Visualisation is a key component in monitoring since it provides aggregation of data in an easy and human-readable interface. However, since many issues are often unpredictable, it is difficult to put in place the right dashboards or queries a priori. It is important then to focus on the possibility of easy and customizable querying of data: (i) helping the customer in writing meaningful queries, e.g., through domain-specific query languages abstracting the more common functions applied to data [38] or, (ii) enabling different development teams to customise the graphical interface easily [57].

2.2 End-to-end distributed tracing

Distributed tracing tools are developed for complex distributed services and micro-service architectures with the purpose of retrieving end-to-end data and analyse the work-flow of requests through system components. Indeed, traditional monitoring tools fail to provide a view of complex interactions between services. On the one hand, *black-boxes approaches* exploiting pre-existing logs cannot wholly reconstruct causality relationships and require many data and expensive processing. On the other hand, monitoring through metrics offers an orthogonal perspective evaluating single service status and performances, but cannot offer any insight into the history of single requests. Monitoring the system through end-to-end trace data, instead, simplifies maintenance and debugging and enables easy detection of critical paths and bottlenecks to optimise performances.

Critical requirements in designing a distributed tracing tool are [38, 57]:

[7] Kubernetes https://kubernetes.io
[8] Grafana https://grafana.com
[9] Kibana https://www.elastic.co/products/kibana
[10] Chronograf https://www.influxdata.com/time-series-platform/chronograf/

- *Low overhead*: collection and retrieval of traces do not have to affect performances of the system and have to be done transparently with respect to system primary activities;

- *Minimal instrumentation required*: to effectively capture interactions between components within work-flows, the vast majority of end-to-end distributed tracing systems adopts ad-hoc instrumentation for *metadata propagation*, i.e., metadata identifying the requests are stored within processes and are transmitted in inter-component communications. Tracing tools must provide easy integration in existing systems without requiring complex modifications for instrumentation.

Two main models are used for end-to-end tracing systems to represent sampled data: the *span model* (see Figure 2.2), adopted for example by Google *Dapper* [57] and the OpenTracing specification [35], and the more general *event model*, adopted for example by X-Trace [31].

As pointed out in [44] the two models presents a trade-off between *simplicity*, in implementing/inspecting data sampled, and *expressiveness*. Moreover, as discussed by *Leavitt*, analysing in details the two alternatives it is possible to prove that *spans* are less powerful model with respect to *events*: each *span* may be defined as a composition of *events* but *spans* may not represent each possible composition of *events*.

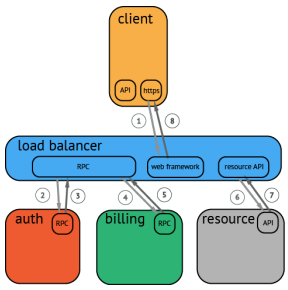

Fig. 2.1 An example of distributed computation. Figure taken from the OpenTracing website[11].

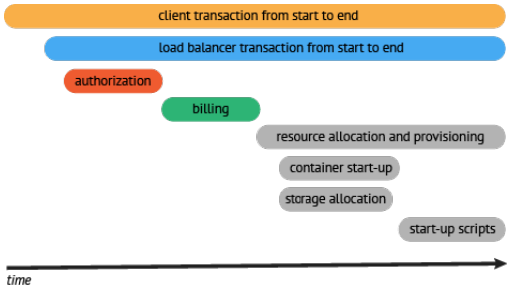

Fig. 2.2 *Span* based representation of the computation aside. Figure taken from the OpenTracing website[11].

In this section, we discuss the main distributed tracing tools presented in the literature and currently used in the industry. To conclude, in Section 2.2.7, we present a comparison table between different data models for trace data.

[11] OpenTracing https://opentracing.io

2.2.1 X-Trace

X-Trace framework is one of the first attempts to trace requests in a networked system preferring a *task*-centric approach to *device*-centric monitoring. X-Trace propagates metadata through the request path by mean of code instrumentation. Each node receiving metadata issues a report and then all reports are collected to reconstruct the trace model.

X-Trace's first model, presented in [31], only defines the concept of *events* and organises them in a tree structure with *edges* indicating a causality relation. Edges may be of two kinds depending on the primitive implemented to propagate metadata: `pushDown()` if the next hop in computation is on a lower software layer and `pushNext()` if it is on the same layer.

The model used by X-Trace is refined in [32] to allow more expressiveness and generality in the representation of computations. The second version of the framework models each trace as a direct acyclic graph (DAG) where edges represent the *happens-before* relation as defined by Lamport [42].

X-Trace collects traces not only at node boundaries but also within a node when the request goes across different software layers. When reconstructing a graph from reports, a *transitive reduction* is operated to determine redundant edges useful to identify subgraphs and to summarise task structures. As proved in [44] this model can represent each type of computation. X-Trace project is no longer maintained, and it has been superseded by the *Pivot Tracing* project (see Section ß2.2.4).

2.2.2 Dapper

Dapper is the first large-scale distributed tracing infrastructure, described in a technical report from Google in 2010 [57]. The model used by *Dapper* for collected data builds on the concepts of trees, spans and annotations.

- Each *tree* represents a computation, i.e., the work-flow of the request through the system.
- *Spans* are the tree nodes and represents a basic block of the computation.
- *Edges* indicates a causal relationship between spans, identifying a hierarchical structure between them. Each edge connects a span and its parent span. A span without a parent identifies a *root span*.
- *Annotations* are additional data recorded with span data. Developers of a specific service can define them, and they may be simple text annotations or key-value pairs. They enable common variables propagation along a request and within a process.

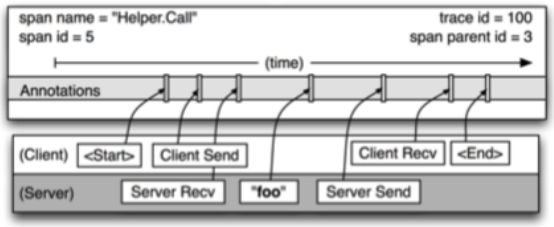

Fig. 2.3 A single span in *Dapper*. Figure taken from [57]

A tiny *trace context* containing trace and span ids is propagated over threads, RPCs calls and over all other inter-process communication (such as HTTP calls or database queries) to enable trace reconstruction. Software engineers at Google exploited the uniformity of threading models, control flow and RPC system used in their systems to build a tracing infrastructure that requires instrumentation of only a small set of shared libraries to propagate trace context and effectively achieve application transparency and pervasiveness.

Each process writes span data to local log files. Those data are then **pulled** from different hosts by *Dapper* daemons that collect and write them to *Dapper* repositories. *Dapper* exploits a sparse representation where each trace represents a row of a table, and each column corresponds to a span. See Figure 2.4.

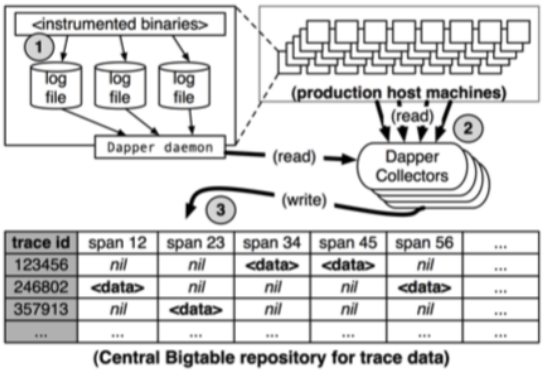

Fig. 2.4 The pipeline in *Dapper*. Figure taken from [57]

Dapper allows analysing data through an interactive web-based user interface providing graphical visualisations of data, such as call tree diagram or traces' metrics trends over time, and allowing analysis on aggregated data (on patterns but also on single traces). The interface is also capable of directly communicate with daemons on production hosts to retrieve real-time data. *Dapper* also offers an API (*DAPI*) for raw trace data in repositories in order to allow developers to access them through multiple analysis tools, already provided or easily implementable.

To reduce overhead due to management on local disks of traces, especially on latency-sensitive services, *Dapper* performs sampling of traces. In the experience reported in [57], aggressive sampling does not affect the reliability of data retrieved for high-throughput services. However, an adaptive sampling strategy, e.g. parametrised by the desired rate of sampled traces per unit of time, is suggested to cope with unstable situations.

One of the primary limits of *Dapper* infrastructure is the low expressiveness of its data model. *Dapper*'s model covers only a small portion of inter-process communication patterns, in particular, multiple-parent relationships (events that cannot occur unless more than one other event has already taken place) are not allowed in trees, and cannot represent exhaustively asynchronous relationships.

2.2.3 OpenTracing

OpenTracing[12] is an open standard for distributed tracing defining a common API, supporting main platforms and languages, to instrument applications and OSS packages in order to enable vendor-neutral trace data collection. The API concepts and terminologies are based on *Dapper* [57] tool experience, but offer more expressiveness.

In OpenTracing, *Dapper* trees are replaced with generic directed acyclic graphs (DAG) of *spans* (allowing for example multi-parent relationships), and edges between *spans* are called *references*. OpenTracing allows two kind of *references*: *ChildOf*, if parent span depends on the child span in some capacity and *FollowsFrom*, if the parent does not depend on the result of the child. Both reference types model direct causal relationships between parent and child spans and in current specification non-causal relationships (e.g. queues) are not supported. *Dapper annotations* are called *span tags*, but OpenTracing distinguishes between *span tags*, information related to the entire span and *span logs*, timestamped events occurring during a span.

A *span context* is used for propagation of metadata across process boundaries. In relation to *span contexts* OpenTracing introduces also *baggage items*. *Baggage items* are *key-value* string pairs propagated within the context, they enable the costless possibility to transmit data over requests paths but must be used with care since baggage transmission may affect the network and CPU overhead.

The OpenTracing standard also proposes *semantic data conventions* prescribing well-known tag names and log fields for common scenarios. Usage of those names in instrumentation is recommended to ensure well-defined data portable across different tracing backends.

[12] OpenTracing https://opentracing.io

Vendor-neutrality allows using the OpenTracing API with a broad set of *Tracer*s, i.e., components to retrieve, collect and manage trace data collected. We analyse the two mostly adopted open-source solutions: *Zipkin* and *Jaeger*.

Zipkin

Zipkin[13] was the first open-source and production-ready clone of *Dapper*. It offers a set of instrumentation libraries, a Java backend collecting traces from instrumented reporters and a complete pipeline to store, query and visualise traces.

Instrumentation libraries are responsible for propagating metadata needed for tracing, for generating trace spans but also for asynchronously sending traces to the backend (possible transports are HTTP, Kafka and Scribe).

Zipkin data model has been designed before OpenTracing specification was defined, so it does not perfectly match it (it is more similar to *Dapper*'s model). Nevertheless, instrumentation libraries have been adapted to be compliant with the OpenTracing standard and to bound the two models.

Jaeger

Jaeger is the end-to-end distributed tracing solution developed by Uber engineers and then released as an open-source project hosted by Cloud Native Computing Foundation (CNCF). Uber solution stems from a need to gain visibility on the company's micro-service based system and its complex interactions without being forced to change already adopted technological solutions.

Uber developed *TChannel*[14], a network multiplexing and framing protocol for RPC, designed to guarantee *Dapper*-like tracing features. However, the Uber infrastructure presented different solutions in managing interprocess communication, so client libraries have been build in different languages to allow instrumentation of all services over the OpenTracing API.

To decouple the collection of trace data from applications, a `jaeger-agent` sidecar process is deployed on every host being in charge of communicating with the tracing backend. It *pulls* the backend for the sampling strategy (allowing for dynamic values of the sampling rate, e.g., determined by the actual load) and it *pushes* trace data exposed by the `jaeger-agent` to the `jaeger-collector`. Client libraries are designed to report trace data to a local UDP port and poll the local `jaeger-agent` to obtain the sampling strategy.

[13] Zipkin https://zipkin.io
[14] TChannel https://github.com/uber/tchannel

Trace data are exchanged between components through a Thrift[15] model based on the OpenTracing specification. *Jaeger* offers a web-based interface to analyse trace data. Traces can be queried and inspected and also a dependency graph can be reconstructed from the interactions between services within a trace.

Comparison

In Table 2.1 we propose a comparison between the two end-to-end tracing systems. *Jaeger* is a younger project with respect to *Zipkin*, but it has a strong and active community working on it and a wide adoption thanks to low memory requirements (executing GoLang components requires fewer resources) and a scalable and decoupled design.

	Zipkin	**Jaeger**
Data model	Custom Made compliant to OpenTracing	OpenTracing based
Collection Mechanism	Each instrumentation library is responsible to sent data to the backend	Decoupled in jaeger-client (language specific Tracer) and jaeger-agent (in GoLang, sending data to the backend)
Sampling rate	Fixed	Dynamic
Backend	Java	GoLang
Scalability	More backend instances can be spawned but traffic from instrumented components must be redirected using a load balancer	When more backend instances are spawned, the jaeger-agent can exploit TChannel's service discovery to automatically balance traffic among collectors

Table 2.1 Comparative table of *Jaeger* and *Zipkin*.

OpenCensus

In comparison with OpenTracing, it is meaningful to consider OpenCensus, a project of Google, recently open-sourced. It consists of a set of libraries for different languages aiming to offer a unique API to instrument code for both application-level metrics and distributed trace data. It also offers the possibility to send data gathered to multiple backends providing a set of exporters.

The API specification is not compliant with OpenTracing, but it is interesting the approach towards unique instrumentation. A conjunct effort, also with the ongoing

[15]Thrift https://thrift.apache.org

OpenMetrics[16] initiative to standardise metrics, would be advisable to offer a standardised solution and a unique API for both application-level metrics and trace data.

2.2.4 Pivot Tracing

Pivot Tracing is a framework to dynamically query events in a distributed system [47]. Tuples models events in a streaming-distributed dataset and *Pivot Tracing* allows evaluating relational queries over it. *Pivot Tracing* combines *dynamic instrumentation* to reduce not needed overhead, and *causal tracing* to correlate events on requests' work-flow. A user can install new queries dynamically exploiting *Pivot Tracing* and its architecture:

- *Front-end client libraries*: define tracepoints, allow to write text queries, compile queries to *advice* and submit them to *PT Agents*;
- *Baggage library*: instrumentation library to propagate data through the execution path between tracepoints;
- *PT Agents*: manage dynamic instrumentation installing *advice* at tracepoints, receive commands and send tuples.

Correlation of events is based on the definition of the *happened-before join* operator that joins on Lamport's happened-before relation [42]. It allows defining queries for arbitrary metrics, predicating about events correlated in the computation flow but happening at different parts of the system.

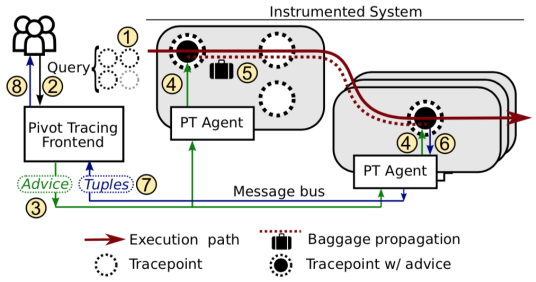

Fig. 2.5 *Pivot Tracing* overview. Figure taken from [47]

In [47] it is also present an analysis of use cases and a detailed report to show how causality within request work-flows allows debugging an issue on the replica selections operated by HDFS (distributed file system used in the Hadoop stack [17]). The debug activity

[16] OpenMetrics.io https://openmetrics.io
[17] Hadoop https://hadoop.apache.org

exploits the *happens-before join* to compare, for each request, block-locations returned by the *NameNode* component and the actual replica chosen by the *client* to retrieve data.

An important issue about *Pivot Tracing* is that it can cause high overhead in the system. Installing complex queries, a not negligible overhead due to baggage may be caused by propagating data over requests flow. *Pivot Tracing* does not operate sampling and it does not apply to static analysis since it gathers data only if queries are installed.

Research work made on *Pivot Tracing* is important to show the effectiveness in debugging distributed systems (i) applying dynamic processing of trace data, and (ii) exploiting in-traces causality relations.

2.2.5 Canopy

Canopy is the Facebook solution to retrieve data about end-to-end execution path of requests [38]. Differently, from others distributed tracing systems, it provides a more comprehensive solution to query and analyse performance data, focusing both on dynamic and static analysis.

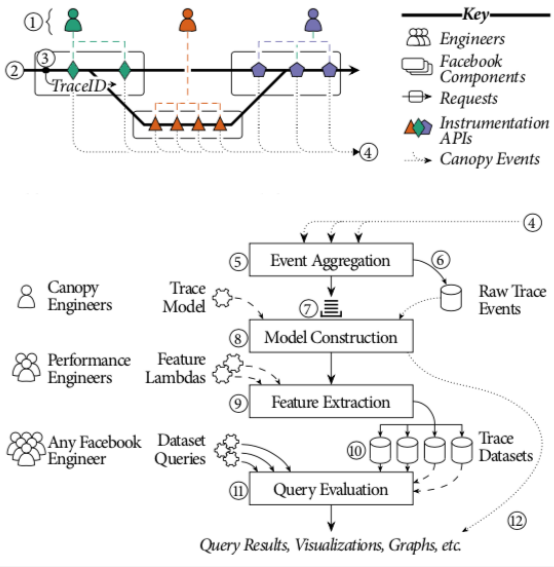

Fig. 2.6 Flow of trace data in *Canopy*. Figure taken from [38].

Canopy adopt various abstractions at different levels to cope with the heterogeneity of its infrastructure and to exploit decoupling. A brief description of *Canopy*'s pipeline:

1. A wide range of **instrumentation APIs** is provided to different services to collect performance data, all of them mapped to a common *Thrift* definition of **events**. Events are related to a *TraceID* (a unique number identifying a request propagated by mean

of instrumentation), can be of different types and are distinguished through ids, a timestamp and an optional sequence number. Events definition also provides the possibility to add *annotations*, key-value pairs containing performance data and information useful to help *Canopy*'s backend to manage events and their dependencies in the graph. APIs can expose high-level primitives to simplify developer instrumentation simply binding behaviours to event types that determine how the backend will interpret them.

2. *Canopy* backend aggregates events by *TraceID* and constructs a **trace model**, a high-level representation of performance traces built analysing edges in the events graph, events types and annotations.

3. **Feature lambdas** are applied to each modelled trace to obtain from raw data predefined high-level features that can speed up common queries. Definition of features may be done on a *per-dataset* or *per-query* level. A domain-specific language (DSL) is provided to describe features as pipelines of functions, but *Canopy* also supports *iPython* notebooks to specify them.

4. **Trace derived datasets** contain different levels of abstraction and features can be related to elements of different granularity (e.g., traces or more specific components within a trace). Engineers can query these datasets but also directly raw data.

Model used by *Canopy* merges both the *Event* and *Span* paradigms and it is similar to the one proposed in [44]. Each trace is represented as a generic DAG of events mapped to a model composed of different components (as defined in [38]): (i) *Execution units* represent high-level computational tasks approximately equivalent to a thread of execution, (ii) *Blocks* represent segments of computation within an execution unit, (iii) *Points* represent instantaneous occurrences of events within a block, and (iv) *Edges* represent non-obvious causal relationships between points. All elements of the model can be annotated with performance data retrieved by events annotations.

Canopy's model provides a common abstraction for traces to hide lower level heterogeneity and since it is decoupled from instrumentation can be updated if necessary.

Canopy does not apply to real-time processing, but highlights the importance and effectiveness of combining dynamic and static analysis to process trace data.

2.2.6 Auto-tracing and service mesh

Tracing tools described can provide value in debugging distributed systems, but often their adoption is limited because of the effort needed in instrumenting software components.

To solve this problem, an emerging approach exploits service meshes to operate some sort of *auto-tracing*.

Service meshes can be constructed deploying in each node lightweight proxies, like the *EnvoyProxy*[18], managing all inbound and outbound networking traffic of the component. Central management of proxies automates tasks like service discovery, routing, health checking, load balancing and authentication and authorisation. Moreover, it allows to report per-request statistics, and it can enable distributed tracing. Indeed, instrumenting only proxies it is possible to support each type and language of software components and propagate metadata to reconstruct traces and interactions in work-flows.

However, despite guaranteeing fast implementation, this method has a significant drawback related to the fact that it considers components as interacting black-boxes. It is nevertheless necessary to instrument single components to provide application-level data essential to enable white-box monitoring.

2.2.7 Trace data models

In Table 2.2, we summarise the principal models used for trace data in the tools presented, and we compare them highlighting advantages and disadvantages of each solution.

Table 2.2 Comparative table of trace data models.

Model	*Elements*	*Advantages*	*Disadvantages*
Events model X-Trace	*Generic DAG* **Event**: single point in time in the computation **Edges**: represent happens-before relation	Generic DAGs can be represented (e.g events with multiple incoming edges). **All kinds of computations can be represented** with this model.	The **model** is **not** so **intuitive** for programmers (both in the instrumentation and analysis phases). Graph reconstruction must be done properly to highlight concurrent structure of tasks in a trace.
Spans model Dapper Zipkin	*Tree* **Spans**: block of computation **Edges**: represent *activation* relation between parent and children spans **Annotations**	**Simplicity** of instrumentation. Easily **understandable model** for programmers and performance engineers.	**Low expressiveness**: the model cannot represent all kinds of computations (e.g. problems with *multi-parent* and *asynchronous relationships*).

[18] EnvoyProxy.io https://www.envoyproxy.io/

Spans model Open Tracing Jaeger	*Generic DAG* **Spans**: block of computation **References**: represent *causal* relationships (*ChildOf* and *FollowsFrom*) ***Tags*** and ***Logs*** ***Baggage***: key-value pairs propagated through request path by mean of tracing instrumentation.	Allows **multi-parent relationships**. **Baggage** offers a costless powerful method to exploit tracing instrumentation for different purposes. **Open source** standardized model and **semantic conventions** on logs e tags.	**Only direct causal relationships**: the current model cannot represent *non-causal* relationships (e.g. queues)
Hybrid model Canopy	*Generic DAG* **Units**: high level computational tasks **Blocks**: segments of computation within an execution unit **Points**: instantaneous occurrences of events within a block **Edges**: non-obvious causal relationships between points **Performance Data**	It **combines advantages** of both the *spans* and the *events models*: interpretability of the model and possibility to represent any computation (e.g also queues, asynchronous executions, multi-parent causality)	Quite **complex model**: mapping from instrumentation-data to the model is not trivial and also an abstraction layer (e.g. features) is needed to help analyzing efficiently modeled trace.

2.3 Information Flow Processing

In this section, we describe a set of tools supporting scenarios that require to process a large amount of data with low latencies. Requirements for real-time stream processing [60] are shared by a large number of distributed applications, for example, systems exploiting sensor networks or monitoring financial trading, and traditional DBMSs[19] cannot address them.

The *Information Flow Processing* (IFP) domain includes systems able to collect information flows from multiple distributed sources and to process them effectively in a timely fashion. This set of systems are denominated IFP Engines and a general framework to characterise and compare them is presented in [17].

Cugola and Margara describe a general model for an IFP Engine: a rule-based system processing heterogeneous information flows to produce new knowledge consumed by other systems (shown in Figure 2.7). In particular, the main components depicted are: (i) a set of ***sources*** generating information at system periphery, (ii) information items flowing to the IFP engine without any guarantees on their ordering or data semantics, (iii) a set of ***processing rules*** specifying how to filter, combine, and aggregate the different flows of

[19] Database Management Systems

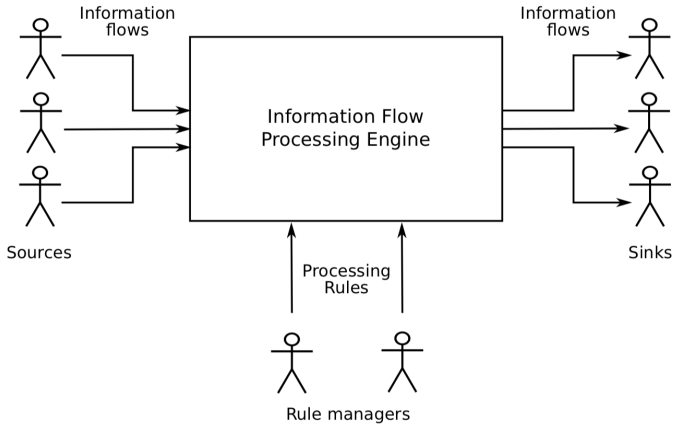

Fig. 2.7 High level architecture of an *Information Flow Processing* engine. Figure taken from [17].

information, (iv) ***processing managers*** entities responsible of adding or removing rules, (v) new flows of information produced as the output of the engine processing, (vi) a set of ***information sinks*** receiving the generated flows.

Information Flow Processing systems reverse the interaction model of traditional databases. DBMSs store data received and can be interrogated through once-executed queries, IFP systems instead offer a continuous evaluation of rules, expressed through graphs of primitive operators, on flowing information items.

Data processed by IFP engines are often related to events in time, and at least a time-based relation is established while processing the flows. Therefore, items are annotated with timestamps, to express rules predicating also on their timings relations, and, for this purpose, two different time domains can be taken into account [2]:

- **Event Time**: time of occurrence of the item or time of occurrence of the event represented by the item,
- **Processing Time**: time at which the processing engine observes the item.

Requirements for an IFP engine are: *expressiveness* to define effective rules, *scalability* to handle large volumes at high rate and guarantee fast responses, and *flexibility* to handle heterogeneous, incorrect or incomplete data.

Different solutions have been implemented in the *IFP domain*, based on: (i) different data models, (ii) different languages to express processing rules, and (iii) different processing techniques. However, they can be generally classified as **Data Stream Management Systems** (DSMS) [7], developed by the database community, or **Complex Event**

Processing (CEP) systems [46], developed by researchers with different backgrounds but interested in event-based systems.

2.3.1 Data Stream Management System

DSMS systems inherit from traditional databases both the data model and the query model. They are developed as an evolution of traditional data processing applying the relational model and exploiting declarative languages derived from SQL.

Issues in applying DBMSs to streams are related to: (i) *unboundedness* of data not representable through finite tables, (ii) impossibility to make assumptions on *data arrival order and rate*, (iii) necessity to apply *one-time processing* avoiding storage to cope with sizing and timing constraints.

DSMSs solve this issue with the concept of *relational data stream*, modelling incoming data, and the *windowing* mechanism to deal with finite portions of data. Therefore, DSMSs focuses on transient data management and allow to run continuous queries which are continuously answered as new data arrives.

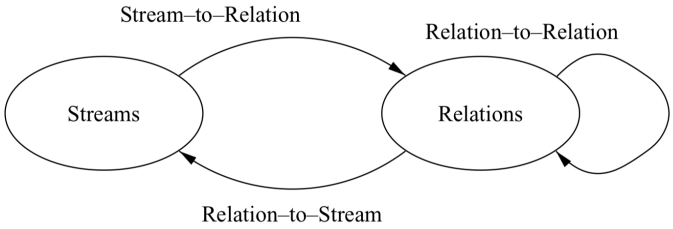

Fig. 2.8 CQL DSMS model proposed in [5].

The CQL stream processing model, proposed by *Arasu et al.* in [5], defines a generic DSMS through three classes of operators (as shown in Figure 2.8):

- the **Stream-to-Relation** (S2R) operators transform the Relational Data Stream, that a priori should be considered unbounded, into a relation, i.e., a finite but time-varying set of tuples. Different operators can be applied, but the usual one is the *sliding window* reporting at any time a finite portion of the stream that is modified (*slides*) over time. Windows' sliding can be *time-based* if based on time-constraints or *tuple-based* if based on the number of tuples (rows) in the window. Moreover, two parameters regulate how the window slides: the width ω, i.e., the time range/number of tuples that are considered to fill each window, and the slide β, i.e., how much time/how many tuples the window moves ahead when it slides.

- The **Relation-to-Relation** (R2R) operators applies relational algebra to the existing time-varying relations in order to produce a relation from one or more other relations.

- The **Relation-to-Stream** (R2S) operators are necessary to output query results as a stream. Every time the continuous query is evaluated, results obtained are processed by the R2S operator and results are appended to the output stream. There are usually three R2S operators: (i) **RStream** outputs as a stream the whole timestamped result of the query each time it is evaluated, (ii) **IStream** outputs as a stream only the new timestamped results, i.e., the difference between the set of tuples computed at the last step and the ones computed at the previous step, and (iii) **DStream** does the opposite of *IStream*, i.e., outputs as a stream the difference between the timestamped results computed at the previous step and the ones computed at the last step.

The main advantage in shifting from traditional processing to a DSMS tool is the familiarity with the relational model. On the other hand, the main drawback is related to the possibility of processing only within portions identified by windows making challenging to express and capture complex temporal patterns.

2.3.2 Complex Event Processing

Complex Event Processing systems exploit the notion of event and associate a precise semantic to the items processed. They consider items as timestamped notifications of events occurrences observed by sources. The first CEP systems have been developed within the context of *publish-subscribe* tools [28]. These systems process incoming information items one event at the time, they filter them through a set of predefined topic or analysing their content, and then forwards events considering subscriptions of each subscriber.

CEP engines extend this behaviour allowing to express rules to detect from the *primitive events*, i.e., the items of the incoming information flow, a set of *composite events*, i.e. representing higher-level events. Indeed, CEP languages, e.g. TESLA [18] enable definition of complex patterns combining rules on events *conjunction*, events *disjunction*, *sequences* of events or *repetition* of events.

Figure 2.9 shows the CEP model proposed in [17]. A CEP Engine is often composed by a network of *event processing agents* composing the *event processing network* responsible for processing events, detecting higher-level information, and notifying sinks subscribed both

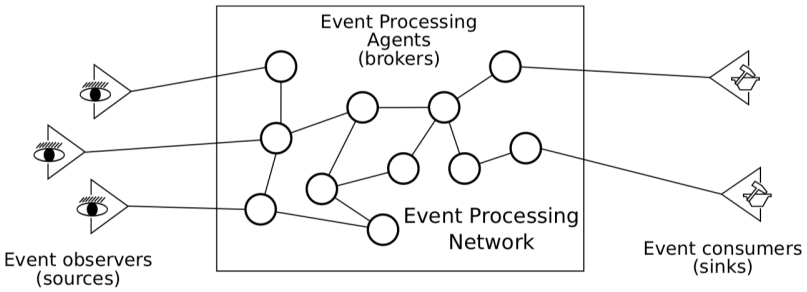

Fig. 2.9 CEP model proposed in [17].

to primitive and composite events. CEP engines are often highly distributed to reduce the overhead on the network and to process and filter events efficiently.

CEP Engines provides a solution to the main limitation of most DSMSs, i.e., the ability to express *detecting rules* addressing complex patterns on the timing relations of items. However, they often do not offer the same expressivity in defining *transforming rules*, i.e., rules manipulating items through a set of operators, better addressed by DSMSs declarative languages.

2.4 Stream Reasoning

Applications requiring a stream-based approach needs on-the-fly processing, low latency responses and capability of managing high volumes of data often also at a high rate. DSMS and CEP systems fulfil these requirements providing timely processing of data but show their limits when data are heterogeneous, have complex domain models or need to combine rich background knowledge to be processed. As highlighted in [25], facing these scenarios with an IFP engine requires to the user *to put large manual effort in developing complex networks of queries*.

The situation described is common in many applications that need to process data presenting *heterogeneity* both at a structural and at a semantic level, i.e. data with different formats and different information "encodings", and that needs to extract higher-level findings in complex domains from low-level data. This set of issues have been addresses from research on *Knowledge Representation* (KR) and *Semantic Web* (SemWeb) providing ways to encode semantic of information in data and to enable inference processing to *reason* on those data. However, these solutions, have been developed for static analysis of data with low volume and frequency.

To bridge the gap and to study how to *reason upon rapidly changing information* [22] a new research area called *stream reasoning* has been devised to integrate stream processing tools and reasoning systems and to fulfil their requirements simultaneously.

In this section, we first overview the *Semantic Web* technologies, relying on *Knowledge Representation* methodologies, and then we discuss technologies developed to address the *stream reasoning* problem.

2.4.1 Semantic Web

The *Semantic Web* is a framework thought to make data available in the World Wide Web shareable and reusable. It involves a set of technologies to provide information with a well-defined meaning that can be understood and also processed from automated agents. Due to the complexity of the concept, the *World Wide Web Consortium* (W3C) proposed a layered approach, presented in Figure 2.10, in which each level contains a set of specifications for semantic technologies.

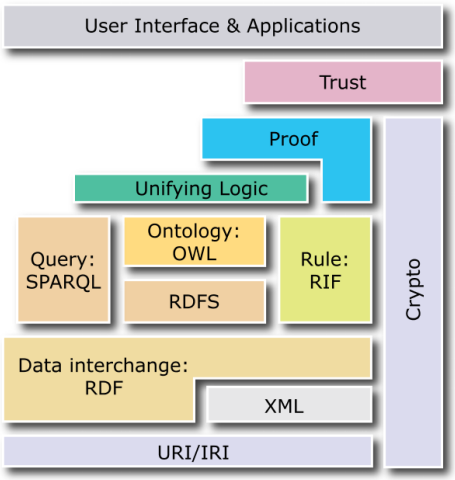

Fig. 2.10 The *semantic web* stack.

In particular, the *semantic of information* is encoded through the *Resource Description Framework* (RDF) that can be used to publish semantically enriched information on the web. Moreover, a combination of the RDF data model and the *Web Ontology Language* (OWL) can enable *reasoning* on exposed data.

Resource Description Framework (RDF)

The Resource Description Framework (RDF) is the W3C specification for data interchange and information representation for the *Semantic Web* [15]. Information is organised in

sets of statements in the form of triples **subject-predicate-object**. Elements in triples can be: (i) *IRI*s (Internationalized Resource Identifiers), a set of characters identifying a generic resource uniquely and allowing to enable equality checking between nodes of different RDF graphs, (ii) *literals*, representing values and represented by character strings associated with an IRI identifying their datatype, and (iii) *blank nodes*, representing anonymous resources and disjoint from IRIs and literals.

RDF triples can be naturally represented as graphs: subjects and objects are the graph nodes, and proprieties are edges connecting nodes.

The Resource Description Framework is a data model and, therefore, it has several possible representations and serialisation formats. The main syntax for RDF models is **RDF/XML**[20], standardized from W3C, defines an XML syntax for describing RDF. Another important format is **JSON-LD**[21], a JSON based serialisation format allowing to exploit the RDF format in systems already using JSON.

W3C also standardizes the **RDF Schema** (RDFS)[22] a data-modelling vocabulary for RDF data. It provides a way to describe groups of related resources and the relationships between these resources.

Web Ontology Language (OWL)

The Web Ontology Language (OWL) is the ontology language for the *semantic web*, it has been standardised by the W3C in 2004 and then updated as OWL2[23] in 2012. It allows representing ontologies, that are conceptual specifications to model knowledge in specific domains through shared and formalised vocabularies. OWL provides a conceptual model to express axioms on resources and relations represented in RDF graphs.

OWL is built starting from the RDF Schema Language, and it provides classes, properties, individuals, and data values. Ontologies itself can be serialised in RDF graphs in the RDF/XML syntax.

OWL enables *reasoning*, i.e., the possibility to infer implicit knowledge from asserted axioms relying on theoretic semantics of *description logics*. OWL2 is modelled to provide the semantics of *SROIQ*, a fragment of First Order Logic guaranteeing decidability and computational practicability of the reasoning procedure. Moreover, in the trade-off between expressiveness and complexity of languages, different **profiles**[24] have been standardised to reduce expressiveness but to make *reasoning* tractable and scalable.

[20] RDF/XML https://www.w3.org/TR/rdf-syntax-grammar/
[21] JSON-LD 1.1 https://w3c.github.io/json-ld-syntax/
[22] RDFS https://www.w3.org/TR/rdf-schema/
[23] OWL2 https://www.w3.org/TR/owl2-overview/
[24] OWL2 Profiles http://www.w3.org/TR/owl2-profiles/#OWL_2_EL

SPARQL

The SPARQL query language for RDF is the W3C standard[25] defining syntax and semantics to express queries on RDF graphs. SPARQL is defined as a graph-matching query language and allows to express queries through the specification of required and optional graph patterns, their conjunction or their disjunction. It formalises an algebra for operators, and it allows to output both results sets or RDF graphs.

SPARQL specifies four different types of queries, each of which can take a WHERE block to specify graph-matching patterns: (i) **SELECT** query returns a set of variables and their matching values, (ii) **CONSTRUCT** query allows returning RDF graphs created directly from query results, (iii) **ASK** query returns a boolean value testing whether or not the query pattern has a solution, (iv) **DESCRIBE** query allows obtaining an RDF graph containing RDF data about the retrieved resources.

The most common query is the SELECT query that can be composed of five clauses: (i) *PREFIX* keyword allows associating a prefix label to an IRI within the query, (ii) *SELECT* keyword specifies variables to be returned and their formats, (iii) *FROM* keyword allows specifying the RDF dataset to query, (iv) *WHERE* keyword provides the graph pattern to be matched against the data graph (simple graph patterns, group patterns, optional patterns, union patterns, filtering patterns, alternative graph patterns), and (v) *Solution modifiers* keywords like *ORDER BY*, *LIMIT*,... allows to modify the result of the query.

The last version is SPARQL 1.1 that extends the 1.0 version with additional features, like aggregations and subqueries. SPARQL queries can be executed on explicitly given graph structures, or can work under some **entailment regime**[26] extending SPARQL semantics to take into account also inferable RDF statements given an *entailment relation* (e.g., RDF entailment, RDFS entailment, etc ...).

Reasoning

Given an RDF graph modelling axioms and a related OWL ontology, it is possible through a *reasoner* component to materialise the graph of implicit knowledge, i.e., the set of relations that can be inferred from the ontology and the individuals (instances of ontology classes) represented in the graph.

Enabling reasoning procedures can provide: (i) *automatic consistency checking* of information present in the graph with respect to the domain modelled by the ontology, (ii) *classification of new information* exploiting also higher level abstractions, (iii) *enrich-*

[25] SPARQL https://www.w3.org/TR/rdf-sparql-query/
[26] Entailment Regimes https://www.w3.org/TR/sparql11-entailment/

ment of query answers with new knowledge inferred from combined graphs, e.g., different sources or historical data.

The main problem of reasoning is its computational cost. Reasoning procedures can become really expensive or even undecidable and, therefore, they have been applied initially only to static data. However, balancing expressiveness, as done with OWL profiles, and designing specific tools, performances can be optimised in order to apply *reasoning* also on dynamic data.

2.4.2 RDF Stream Processing (RSP)

Stream reasoning investigates *"how to perform online logical reasoning over highly dynamic data"* [25]. In particular, we focus on research made on RDF Stream Processing (RSP) to design continuous query models and languages capable of supporting prototypes extending SPARQL to process RDF streams.

RSP-QL

The RSP-QL model, proposed by *Dell'Aglio et al.* in [24], aims at providing a unifying semantic for RSP engines and at describing existing continuous SPARQL extensions and their operational semantics. This model proposes semantics similar to the one used in DSMSs [5].

RSP-QL identifies **RDF streams** as pairs (G_i, t_i), where t_i is the timestamp and G_i is a named RDF graph. Given an RDF Stream, the time-based sliding window operator $\mathbb{W}$ defines a *Time-Varying Graph* that is a function returning for each time instant an *Instantaneous RDF Graph*. Each window is characterized by three parameters, the starting time t_0, the window width α and the sliding parameter β.

An RSP-QL query is continuously evaluated against a *streaming dataset* (SDS) composed by: (i) an optional default graph, (ii) zero or more named graphs and, (iii) zero or more named time-varying graphs obtained applying the sliding window operators over a number of streams.

Evaluation time instants (ET) are defined by mean of reporting policies: (i) **CC** (Content Change) if the window content changes, (ii) **WC** (Window Close) if the current window closes, (iii) **NC** (Non-empty Content) if the current window is not empty, and (iv) **P** (Periodic) at regular intervals.

Outputs of an RSP-QL query may be either a sequence of solution mappings, i.e., a sequence of compliant SPARQL answers, or a sequence of timestamped solutions mappings if a streaming operator is applied. Streaming operators, *RStream*, *IStream* and *DStream*,

are defined similarly to the R2S operators in [5] and append at each evaluation a new set of elements to the output stream with respect to the logic they implement.

RSP Engine

The RSP-QL model can be applied to explain the operational semantics of popular RSP Engines: the C-SPARQL Engine [9], CQELS [43], and Morph$_{stream}$ [13]. Moreover, it can describe the semantics of the different continuous extensions of SPARQL proposed by these engines, respectively, C-SPARQL, CQELS-QL and SPARQL$_{stream}$.

All the three engines consider a particular case where the RDF stream is made by timestamped graphs composed by only one triple, i.e., they assume timestamped RDF triples instead of timestamped graphs. **C-SPARQL** engine exploits a DSMS to operate windowing on incoming data. Once retrieved the *Instantaneous RDF Graph* the query is carried out by a SPARQL engine. **Morph**$_{stream}$, instead, exploits virtual RDF streams and a mapping language to translate queries written in SPARQL$_{stream}$ into several DSMSs queries. To conclude, **CQELS** engine integrates both the windowing mechanism into a SPARQL engine and also implements techniques to manage performances with respect to the velocity of the incoming stream.

The main limitation on the expressiveness of this systems is related to their similarity to the DSMS approach. They show difficulties in capturing complex temporal patterns and, for this reason, an approach towards *Ontology Based Event Processing* has been proposed by *Tommasini et al.* in [62].

Inference Process

In the *stream reasoning* context, processing of streams requires a critical design choice about the moment in which the inference process must be taken into account. This choice can impact both performances and expressiveness and characterises the different solutions proposed to approach the *stream reasoning* problem.

The discussed RSP Engines do not implement, as default, any entailment regime and do not consider ontologies and inferential processing. C-SPARQL and CQELS can operate under RDFS entailment regime, but this deteriorates performances. To face this issue, *Barbieri et al.* proposed in [10] an approach to incrementally maintain the entailed triples (materialisation) avoiding recomputing it from zero at each window change.

2.5 Design Science

In this section, we analyse the fundamental concepts of the *Design Science Methodology* for information systems and software engineering described by Wieringa in [66] and applied in this thesis to develop and discuss *Kaiju* and *Rim* prototypes.

2.5.1 The methodology

Design science is composed by two main activities: ***Design*** and ***Investigation***. The former is related to the design of an artifact to improve a context, and the latter to the investigation of the interactions between the artifact and the context to gain knowledge and to pose new design problems.

Problems addressed by design science are *improvements problems*, i.e., problems posed with the purpose to improve a context. A fundamental claim of the *design science* methodology is related to the fact that the artifact alone doesn't solve any problem and, for this reason, it makes sense and must be designed and studied only in its interactions with a given context. The context can be divided into *social context*, related to stakeholders related to a project, and the *knowledge context*, providing existing designs and answers for the problem addressed. These two contexts are related to two different kinds of research problems: (i) **design problems**, requiring knowledge on the stakeholders and their goals and evaluating different solutions by *utility*, and (ii) **knowledge questions**, requiring familiarity with the knowledge context of the project and evaluating solutions assuming only one *true* answer to be found.

Therefore, stakeholders are more interested in the outcomes of the design activity, while the investigation activity provides answers to improve the understanding in a given *knowledge context*. These activities are tightly coupled, and a *design science* project is based on the continuous alternation of solving *design problems* and answering *knowledge questions*.

Design science is a *middle range science* since it operates under assumptions simplifying *conditions of practice* and focuses on finding generalisation beyond the case level but not universal. Often *design science* projects start from simple simulations of new technologies under idealised conditions. Once evaluated if something is possible at all, idealisations are progressively removed in an iterative process to *scale up new technology*.

2.5.2 Research projects

Research projects are characterised by research goals and by the set of related research problems. Design science projects iterate over designing and investigating activities and, therefore, we can divide projects' research goals into *design goals* and *knowledge goals*.

Design Goals

Design goals define design problems that are mainly related to *technical research questions*. Given a *design goal* it is necessary to identify:

- ***The problem context***: the problem that should be addressed by the artifact and the existing knowledge on the problem to be solved;
- ***The artifact to (re)design***: what should be designed and, if applicable, the existing designs;
- ***The requirements***: what are the required interactions between the artifact and its context and what are the properties the artifact should implement;
- ***The stakeholders and their goals***: who is interested or may be interested in the project and what are their goals.

Problems are usually formulated employing questions and to formulate a *design question* it is sufficient to substitute elicited information in the following template: *"How to <(re)design an artifact> that satisfies <requirements> so that <stakeholders goals can be achieved> in <problem context>?"*. It can be possible not all information are known when starting a project and subsequent iterations can be necessary to define or refine them.

Knowledge Goals

On the other hand, knowledge goals should be refined in knowledge questions, that in case of design science are **empirical**. Indeed, *design science* aims at improvements in a real context, and for this reason, their knowledge questions require data about the world to be answered. Differently from *analytical knowledge questions*, only exploiting conceptual analysis, it is not possible to address them.

Empirical knowledge questions may be classified in different ways. A general classification can be done through a bi-dimensional matrix identifying questions as *descriptive* or *explanatory* on one dimension and as *open* or *closed* questions on the other:

1.a) ***Descriptive questions*** asks for an objective description of what happened without requiring any explanations;

1.b) ***Explanatory questions*** ask causes, effect and reasons of something happened;

2.a) ***Open questions*** does not contain a specification of possible answers;

2.b) ***Closed questions*** contains hypotheses on possible answers.

Another possible classification is, instead, specific to design science research:

- ***Effect questions*** asks for the effects and performances in the interaction between the artifact and its context;
- ***Trade-off questions*** ask for a comparisons between alternatives artifacts, or previous versions of the same artifact, in their interaction with the context;
- ***Sensitivity questions*** asks for evaluation of the artifact if the context changes;
- ***Requirements satisfaction questions*** asks if effects and performances registered satisfy functional and non-functional requirements.

When formulating knowledge questions, it is necessary to distinguish them from **prediction problems**, i.e., questions asking for something in the future. These questions are related to an additional type of problems that should be considered together with design and knowledge problems and not as knowledge questions. However, generalisations derived from answering knowledge questions can be used to solve prediction problems.

To conclude, it is important to highlight the relation between knowledge and design goals. There is always at least one knowledge goal in a research project, and design goals can be hierarchically defined to achieve it. However, it is possible to have also the opposite situation with a knowledge goal helping to solve a design problem related to the artifact.

Chapter 3

Problem Statement

In this chapter, we define the *observability* problem, we discuss its relevance, and we provide the research questions, the hypotheses and a requirement analysis.

3.1 Observability for software

Software is acquiring an essential role in almost any business [4]. Nowadays, more than ever before, companies are concerned in deploying, maintaining and improving software, for products, but also as assets that solve internal problems and help to gain a competitive advantage. Consequently, all aspects related to how to use software in production effectively are getting momentum.

Unpredictable and erroneous software behaviours call for continuous supervision of execution *correctness* and *performance*. Moreover, business analytics are a key aspect to be competitive on the market [21] and inspecting software components interactions, during actual use by internal users or clients, can provide useful insight also from a business intelligence point of view. Therefore, it is valuable to guarantee a prompt answer to the following questions about software: *What can go wrong?* and *What can we do better?*.

Answering these questions properly is harder given the current evolution of software systems. Indeed, the growing popularity of open-source *orchestration systems*, the increasing availability of *lightweight virtualisation techniques* and the *decreasing costs* of on-demand cloud services offered by a wide range of providers, accelerated the development of service-oriented architectures [33]. This modularisation of applications is shifting software systems complexity from components to the intricate network of interactions between them [27, 30]. Furthermore, when modules are run using virtualisation techniques like containers, which are naturally ephemeral, it is hard to observe the system global status and fundamental tasks such as debugging or scaling become harder too.

For these reasons, new methods helping to supervise such systems at runtime are relevant as never before and can benefit the lifecycle of the software products. The problem of answering the aforementioned questions, in the context described, has been referred under the ***observability*** keyword. While it was previously prerogative of a little number of companies owning large-scale systems [64], currently, given the increasing complexity of even small software systems, it has become relevant for many development teams. A large set of industrial tools (APM vendors[1] and logging-intensive solutions[2]) have been proposed to mitigate it, approaching *observability* from a practical and empirical perspective.

Despite the scientific literature on monitoring and debugging of large-scale system [63] and several attempts to approach the topic from a technical point of view [6, 11, 12, 20, 29, 59, 64], it is not possible to find a shared and structured definition of *observability*. However, the magnitude of the problem calls for a more structured approach. Therefore, the first question we focus on is:

> Can we provide a definition of *observability* for software-based systems?

The intuition to answer this question comes from [48] which was also inspired by [39][3]: "***Observability*** *for software engineers 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' intuition is the notion of *behaviour*, i.e., *what is happening* within a distributed system at runtime.

Software systems are black-boxes that receive inputs data, process them with respect to a domain specific logic and produce some sort of application outputs [55]. In a black-box, we can characterise the internal state only in terms of inputs, outputs and application logic (as shown in Figure 3.1).

Therefore, we formulate a definition of *observability for software* systems based on the concept of **system behaviour**:

> **Observability** (for software systems) is the property of a system to expose its behaviour at runtime through its outputs.

[1] *see* 2018 Gartner Magic Quadrant for Application Performance Monitoring (APM) Suites *https://www.dynatrace.com/gartner-magic-quadrant-application-performance-monitoring-suites/*

[2] *e.g.* Honeycomb *https://www.honeycomb.io/*, Humio *https://www.humio.com/*

[3] In the context of Control Theory (CT), *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 [39].

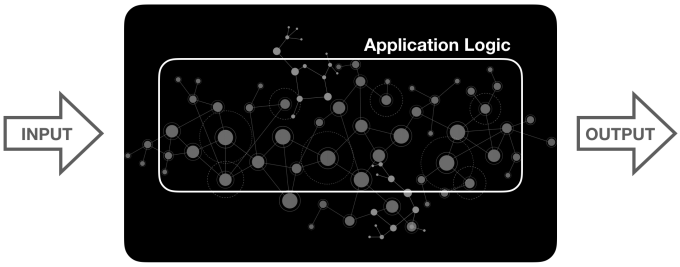

Fig. 3.1 Running software is like a black-box.

3.2 The observability problems

The provided definition of *observability* leads to questions about two tightly coupled but parallel problems:

> **Observability problems** (OP)
>
> **OP1** *How can we expose the system behaviour through the outputs?*
>
> **OP2** *How can we make sense of system behaviour?*

In the following sections, we identify the challenges that should be addressed while approaching these two problems and we elicit the relevant requirements.

3.2.1 Observable behaviour

To approach the **OP1** we need to define data composing the ***observable behaviour*** of the system, i.e., the set of data that can be retrieved as outputs of the system to infer the system behaviour.

The problem to observe a system during its execution has been traditionally addressed by: (i) observing *application outputs* correctness (with respect to inputs and application logic), (ii) collecting and processing *metrics* from system components, (iii) collecting and analyzing *logs* from system components, and (iv) evaluating *traces* of (distributed) computations performed (event-based or state-based debugging [19], in some cases empowered by automatic tools [67]).

Many methodologies evolved during the years to cope with the shift of paradigm from monoliths to distributed architectures and then to micro-service based architectures. On the one hand, agent-based collection mechanisms for logs and metrics, combined with new storage and processing solution, made monitoring scalable [1, 41, 49]. On the

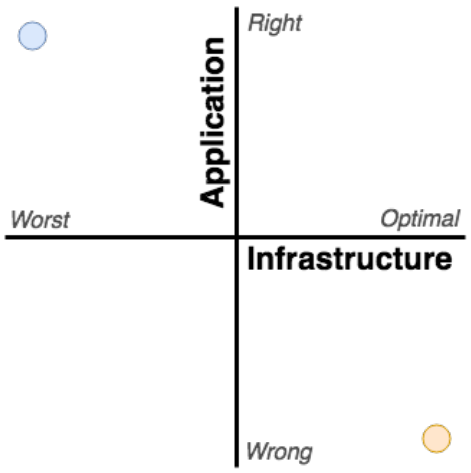

Fig. 3.2 Axes defining the *behaviour* of a software-based system.
The two points in figure represents two extreme and opposite situations. The light-blue point represents a correct behaviour performing really bad, while the orange point represents a performant but wrong behaviour.

other hand, trace collection evolved to adapt to highly distributed settings through the development of end-to-end distributed tracing tools empowering analysis of complex concurrent executions through metadata propagation [38, 57].

Considering the nature of these methods we can classify them as descriptors of two aspects of the *system behaviour at runtime* (depicted in Figure 3.2):

- ***Application***: axis representing the system correctness in implementing the specification functions between its inputs and its outputs;
- ***Infrastructure***: axis representing the system performances in meeting efficiency and availability requirements.

The two dimensions are orthogonal, e.g., a system may perform service operations correctly but with really high latencies making it useless (light-blue point in Figure 3.2); and a system may provide results with low latencies (e.g. as defined by non-functional requirements) but its services can perform wrong operations with respect to the functional requirements (orange point in Figure 3.2).

Given this framework and considering current software systems, we can then claim *application outputs* provides the *observable behaviour* of the system on the *application* axis, while *metrics, logs* and *trace data* do the same on the *infrastructure* axis. Therefore, we define the *observable behaviour*.

> The ***observable behaviour*** is the behaviour subsumed by the all available outputs, i.e.,*application outputs*,*metrics*, *logs*, *trace data*.

The adjective *"observable"* highlights that we can examine only the system's outputs. In this regard, *application outputs* are domain-specific and already defined by functional requirements whereas *metrics*, *logs* and *trace data* instead need ad-hoc instrumentation that must be carefully designed to guarantee meaningful outputs. For this reason, the **OP1** is mainly related to the *infrastructure axis* and to these three types of data, we will name ***observations***.

Indeed, the problem about exposing system behaviour is not only a matter to have this outputs put in place. It is not trivial to determine the right *observations* a system must expose in the trade-off between collecting too many data and not exposing enough information. Therefore, even if determining what should be exposed may be dependent on the specific piece of software, a set of guidelines to deal with this two-sided risk should be provided dealing with **OP1**.

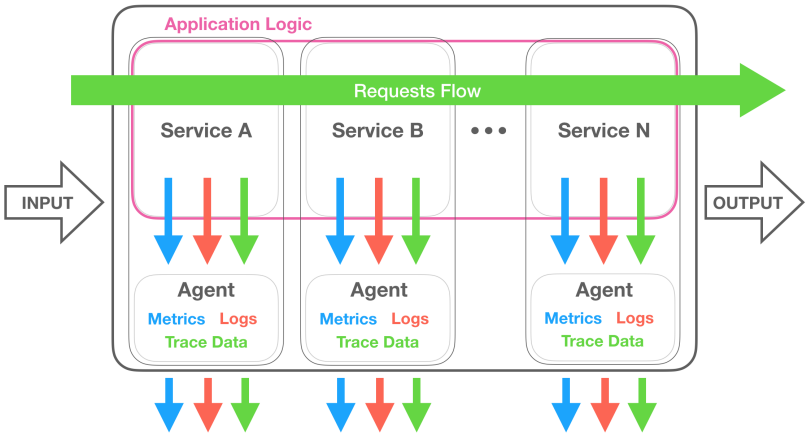

Fig. 3.3 Metrics, logs and trace data collection.
Observations are gathered from different components and then processed as three separated flows.

Currently, different tools exist to deal with metrics, logs and trace data but each tool is often specialised to gather, process and analyse only one of these types of data (Figure 3.3). This differentiation of formats is mainly due to the *complexity of domains* these data are meant to represent. However, as pointed out, all *observations* convey clues on the actual system behaviour: it is not possible to provide a strict categorisation of their data and, on the contrary, their *interoperability* is desirable. Indeed, the definition boundaries of

metrics, logs and trace data are often blurred, and the same information is represented multiple times under multiple formats. It is clear the lack of tools capable of providing a unified instrument avoiding deploying multiple pipelines and allowing to reconcile the different perspectives towards a common solution for the *observability problem*.

Therefore, trying to answer **OP1** we pose a further question:

> **Q1** Is it possible to unify the data models and processing pipelines of metrics, logs and trace data (i.e., *observations*) to provide a single and significative output for the *observable behaviour* of a software system?

To approach this question, trying to define a common data model, we elicit a set of *requirements* the proposed answer should fulfill:

Q1.*R1* Timestamped
Observations describe the behaviour of the system over time and so its information content is strictly bound to a specific instant 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.

Q1.*R2* Flexible Schema
To empower *functional 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 to model the complexity of the domain.

Q1.*R3* Shared Terminology
To empower *semantic interoperability* of *observations* gathered from different components of the system, shared vocabularies are desirable, e.g., for error identifiers, components and operation names, resource naming, metrics naming, etc...

3.2.2 Behaviour interpretation

The **OP2** problem is about the methods to derive insights from what we called the *observable behaviour* of a system.

We start defining an *interpretation of behaviour*:

> An *Interpretation of behaviour*, in the general case, is a function of the *observable behaviour* of the system.

However, *application outputs* do not require pre-processing to provide information and are often addressed within the context of *testing*. For these reasons, also with respect to **OP2** the main challenge is related to *observations*. Therefore, we define the *interpretation of behaviour on the infrastructure axis* as *interpretation of observations*:

> ***Interpretation of observations*** $= f$(Metrics, Logs, Trace Data)

It is important to notice that even if we are interested in the behaviour of the system at runtime, the processing of data needs to be both *dynamic* and *static* to address different use cases related to *observability*. For example, applying machine learning algorithms to set thresholds based on historical data calls for a static analysis of the system behaviour, while detecting increasing latencies or checking SLA[4] requirements at runtime calls for dynamic analysis.

However, there are a number of reasons why dynamic analysis is becoming more relevant. Firstly complex systems fail in really complex ways [16] and, thus, no effective test can be done to ensure coverage of all possible situations and facets a largely distributed system might exhibit. Therefore, testing software in production [48] with a prompt reaction to erroneous behaviours is often the last resort. Second, *observations* [59] are vast and ephemeral: process them *on-the-fly* is the only option since there is not enough time, and often not enough space too, to store and then process.

For these reasons, we focus on making sense of system behaviour through dynamic analysis, without neglecting the necessity to persist data to allow also static analysis. Under this assumptions, we can then pose a further question about **OP2**:

> **Q2** Is it possible to make sense in near real-time of information needs about the system observable behaviour considering the available *observations* at runtime, and despite data heterogeneity?

Similarly to **Q1**, we elicit a set of requirements that a solution should fulfil.

[4] Service Level Agreement

Q2.*R1* Process Temporal Relations

As stated by **Q1.*R1***, the lowest common denominator between *observations* is their dependence on time and so we can factor it in the definition of *interpretation of observations*:

$$\textbf{\textit{Interpretation of observations}} = f(\mathrm{M}(t), \mathrm{L}(t), \mathrm{T}(t)) = f(t,\ \mathrm{M},\ \mathrm{L},\ \mathrm{T})$$

The resulting formula clearly represents the need to interpret *observations* and then process them taking into account their temporal dependencies. The timing of *observations* is fundamental to interpret them with respect to the behaviour of the system. We should be able to express a set of time-dependent rules to process them [17]:

a) *Transforming rules*: we should be able to make continuous queries against observations [5];

b) *Detecting rules*: we should be able to define and detect simple and composite patterns on their content and their temporal relations [14].

Q2.*R2* Reactiveness

The importance of near real-time supervision of systems mainly targets the need to observe and process the current status of the system [8, 58]. We should be able to be reactive, i.e., to observe system behaviour minimizing the processing delay.

Q2.*R3* Handle complexity

The variety of data represented in *observations* and the complexity of domains leads to complex schemas of data. We need to handle this complexity providing a set of operators to effectively *filter*, *aggregate* and *join observations*.

Chapter 4

Design

In this chapter, we address the research questions proposed in Chapter 3.

In Section 4.1 we analyse metrics, logs and trace data models considering current standards. We start pointing out similarities and differences among *observations* data-types, and we show how the *event* abstraction can provide a useful higher-level abstraction reconciling them. To address **Q1** and the elicited requirements, we propose the *observability event* data model and we discuss it with respect to **OP1**.

In Section 4.2, we deal with **OP2** analysing processing models for *observations*. We discuss the need for a stream-oriented approach towards *observability* given **Q2** requirements. Therefore, we explain our choices to validate our claim. We follow the investigation method proposed in [66] modelling two different prototypes: an environment, *Rim*, capable of emulating the context we would like to address and, *Kaiju*, a prototype processing trace data as a stream.

In conclusion, in Section 4.3, we discuss the possibility of crossing the streams processing together different types of *observations*. We explain how stream reasoning can provide a suitable solution for the *observability* problem fulfilling requirements elicited for both questions **Q1** and **Q2**. Therefore, we detail a specification for a prototype processing *observations* through a *RSP-QL* query language [24].

4.1 Modeling observations

As discussed in Chapter 3, *observations* are the three methods currently used to expose the *observable behaviour* of the system on the infrastructure axis. Each one of these provides a correct yet partial view of the system behaviour at runtime. In this section, we discuss **Q1** addressing this problem. In Section 4.1.1 we analyse the most widely accepted standards for metrics, logs and trace data. Taking into account considerations made on these data, in

Section 4.1.2 we propose the *observability event* data model, we detail how the proposed model matches requirements **Q1.*R1***, **Q1.*R2*** and **Q1.*R3*** and we discuss how this model addresses **OP1**.

4.1.1 Metrics, logs and trace data

Metric *A metric is a timestamped and tagged data about system status.* Metrics are collected periodically and can be divided in *work metrics*, related to the performances of running processes, and *resource metrics*, related to the status of physical (e.g. CPU) and virtual (e.g. a queue) resources being used [20]. Since the discussion for an open standard for metrics is still ongoing (Open Metrics[1]), we henceforth consider as a reference the Prometheus format[2], taken as the basis for the standardisation process. Prometheus format for metrics is:

`id` [*metric name* and *labels*] + `sample` [*timestamp* and *value*]

It is composed by two main parts: (i) the *id* part that identifies the metric through a *name* and a set of key-value pairs (*labels*) to provide additional metadata, and (ii) the *sample* part specifying the timestamped value of the metric.

```
1 api_http_requests_total {method="POST", handler="/messages"}    1542650713    34
2 api_http_requests_total {method="POST", handler="/messages"}    1542651425    45
```

Listing 4.1 Example of two samples of the same metric in the Prometheus format.

Log *A log is a time-stamped report of a system performed operation or error encountered.* Logs describe the reported operation/error either in a *unstructured* form, i.e., plain text, or in a *structured* form, i.e., a blob of JSON or a binary serialisation of structured data. Structured data simplifies the automatic processing of logs but, for this purpose, also unstructured payloads are usually formatted following some specification, e.g., RFC 5424[3].

```
1 DEBUG 2018-11-10 16:17:58 - Incoming request from clientId 54732
```

Listing 4.2 Example of plain text unstructured log

[1] Open Metrics https://openmetrics.io
[2] Prometheus: open-source monitoring and alerting system https://prometheus.io/docs
[3] RFC 5424 https://tools.ietf.org/html/rfc5424

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

Listing 4.3 Example of structured log in JSON

Trace data *Trace data are timestamped data collected through requests flows and causally correlated by mean of the happens-before semantic [42].* A trace captures the work performed by a system to process a request. Formats for trace data may be different, as discussed in Section 2.2.7, but we consider the OpenTracing specification [35], a vendor-agnostic effort towards a standardization (detailed in Section 2.2.3). Each span represents a unit of computation within the request workflow. A span is composed by metadata to reconstruct the trace (*spanId*, *traceId* and *references* to other spans), a timestamp representing its *start time*, a timestamp related to its *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 serialisation format for data gathered depends on the specific tracer chosen. For this reason, as exemplified in Listing 4.4, the serialisation format of the spans may be slightly different from the OpenTracing specification.

```
{
    "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"}, {"key": "http.url", "type": "string", "value": "/customer?customer
        =392"}, {"key": "component", "type": "string", "value": "net/http"}, {"key": "http.status_code
        ", "type": "int64", "value": 200}],
    "logs": [{"timestamp": 1542877899033827, "fields": [{"key": "event", "type": "string", "value": "
        HTTP request received"},{"key": "level", "type": "string", "value": "INFO"}, {"key": "method",
         "type": "string", "value": "GET"}, {"key": "url", "type": "string", "value": "/customer?
        customer=392"}]}, {"timestamp": 1542877899033872, "fields": [{"key": "event", "type": "string"
        , "value": "Loading customer"}, {"key": "customer_id", "type": "string", "value": "392"}, {"
        key": "level","type": "string","value": "INFO"}]}]
}
```

Listing 4.4 Example of a span collected with the OpenTracing API and serialized by *Jaeger* tracer.

This three types of data are meant to handle different aspects of the system behaviour and, for this reason, different specialised tools have been built to deal with their specific data formats. The main issue with the current approach is that each type of data is

processed separately. In doing so, we miss the opportunity to join the information-content of differently typed data and to obtain an overall perspective on the status of the system.

4.1.2 Observability events

If we consider in more details the descriptions provided and we try to identify the characterizing feature of each type of data we can classify: (i) metrics as *aggregatable* data given their numeric nature (ii) logs as *records* about software execution, and (iii) trace data as *request-scoped* data.

Given the following categorization of *observations* it is, however, easy to notice we can identify data about system behaviour at the intersection of multiple categories, as shown in Figure 4.1: (i) *aggregatable records*, e.g., logs containing numeric information that is useful to manipulate as metrics data, (ii) *request-scoped records*, e.g., logs related to a span, (iii) *request-scoped metric*, e.g., metrics on resources usage of a request, and (iv) *request-scoped aggregatable records*, e.g., a trace containing request-scoped logs and metrics.

These considerations support our claim of metrics, logs and trace data as a partial view of the same problem. Therefore, to cope effectively with data in overlapping zones, we need to reconcile them avoiding possible replication or split of information.

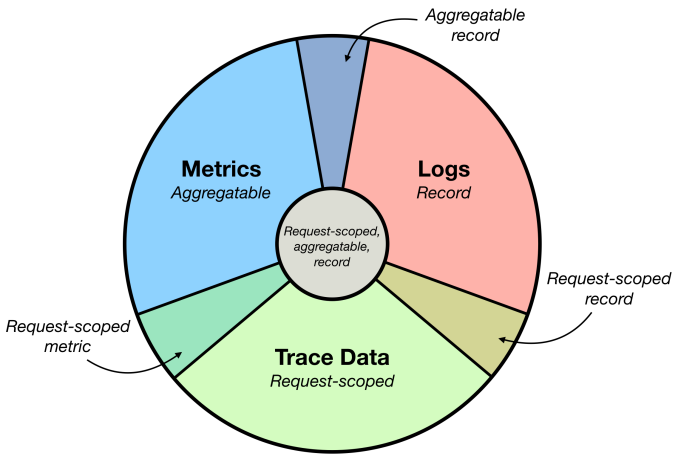

Fig. 4.1 Metrics, logs and trace data are overlapping concepts.
Diagram inspired by [11].

Observations are descriptors of the behaviour of the system at runtime, i.e., of what happens in the system. *Cunha et al.* said: *"An explicit event-based approach seems more adequate, both to describe the dynamic distributed program evolution and to provide a transparent interpretation of correctness properties"* [19]. Therefore, while we consider

useful to have lower-level abstractions, like the concepts of metrics, logs and traces, we claim that approaching ***observations*** as ***events*** can help in providing a higher-level unified point of view towards a single significative output (**Q1**).

Taking into account the actual formats for metrics, logs and trace data and the requirements **Q1.*R1***, **Q1.*R2*** and **Q1.*R3*** we can model the *observability events* as:

$$\textbf{\textit{Observability event}} = \texttt{Timestamp} \mid \texttt{Payload} \mid \texttt{Context}$$

where:

- `timestamp` is a numeric value representing time (e.g., `int64` counting milliseconds since epoch, i.e. 1970-01-01 00:00:00 UTC);
- `payload` is a generic set of key-value pairs containing the actual information content of the event;
- `context` is a generic set of key-value pairs identifying the type of event (i.e., keys in the payload) and providing additional metadata contextualizing the event (e.g., dimensions and scopes of the event like service name, availability zone, instance type, software version, etc...).

We proceed to discuss how this proposed model fulfils **Q1** requirements and how it can be helpful to expose the system behaviour.

Q1.*R1* Timestamped

The common characteristic between *observations* is their dependency on time and also current data formats require a timestamp for each of these data types. In the event world, time is a first-class citizen [46] and, therefore, the proposed *observability event* model requires a `timestamp` as the only mandatory field in its specification.

Q1.*R2* Flexible schema

The proposed model is flexible enough to represent metrics, logs and trace data, to guarantee the reconciliation of their models and the extensibility of formats. Moreover, it also enables expressiveness in complex domains providing a way to make *observability events* multidimensional as modeled by *Pedersen et al.*: payload contains the *facts*, i.e., the actual data (the *"measure"*), and context provides dimensions. Indeed, multiple dimensions provide a way to model complex data: dimensions add axes on which data can be aggregated or sliced and can be used to model hierarchies [53].

We explain how it is possible to bind described data formats to the model proposed and, exploiting a JSON serialisation. Furthermore, we provide examples on this starting from Listings 4.1, 4.3 and 4.4.

Metrics are *observability events* with the `timestamp` of the sampled value, the `payload` composed by the value and the metric name, and `context` composed by the metric labels.

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

```
{
  "timestamp": 1542651425000000,
  "payload": {
    "name": "api_http_requests_total",
    "value": "45"
  },
  "context": {
    "method": "POST",
    "handler": "messages"
  }
}
```

Listing 4.5 Example of metrics as *observability events*.

Logs are *observability events* with the `timestamp` of the record, the `payload` composed by structured key-value messages describing the record, and the `context` providing additional metadata (e.g., the specification to interpret the payload or metadata about the process emitting it).

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

Listing 4.6 Example of a log as *observability event*.

Trace data, despite being more complex than metrics and logs, are *observability events*. A wide range of possibilities can be considered to decompose a trace in *observability events* [38, 44]. Nevertheless, considering the OpenTracing specification we propose to associate one *observability event* to each span. The start time of the span becomes the `timestamp`. The `payload` contains the *end timestamp* (or, without loss of generality, the *duration*), the *operation name* and the *references* to other spans, while the `context` is made by the *traceId*, the *spanId* and the *tags* related to the span (e.g. metadata about the process running the span).

With regard to span *logs* they can be modelled as separated *observability event*s representing logs and containing the *traceId* and *spanId* of the related span in their `context`.

```
{
  "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
  }
},
```

```
{
  "timestamp": 1542877899033827,
  "payload": {
    "msg": {"event":"HTTP request received", "
        method": "GET", "url":"/customer?
        customer=392"},
    "level": "INFO"
  },
  "context": {
    "traceId": "f6c3c9fedb846d5",
    "spanId": "5cfac2ce41efa896"
  }
},
{
  "timestamp": 1542877899033872,
  "payload": {
    "msg": {"event":"Loading customer", "
        customer_id": 392},
    "level": "INFO"
  },
  "context": {
    "traceId": "f6c3c9fedb846d5",
    "spanId": "5cfac2ce41efa896"
  }
}
```

Listing 4.7 Example of a span as *observability events*.

Q1.*R3* Shared terminology

The proposed model enables the possibility to use a shared terminology defining a set of common keys and values. A specification of the terminology is outside the scope of the data model and is not directly enforced by it, but it is supported by mean of a single flexible structure for *observations*.

A unified perspective

It is worth to notice we do not aim to provide a fixed schema for all *observability events*, but rather to offer a unique unified structure for *observations*. Indeed, reasoning on how to expose the system behaviour through a unique data model can provide several benefits:

i) it allows to represent easily *observations* not categorizable as a metric, log or trace data (overlapping zones in Figure 4.1) avoiding replication or split of information under multiple formats;

ii) from a technical point of view it could enable a unique instrumentation API and a unique pipeline to collect *observations* in a distributed fashion (as shown in Figure 4.2);

iii) designing *observations* instrumentation as a whole with a shared terminology can help to avoid replication of information with different semantics;

iv) the context enables the possibility to slice, aggregate and join *observability events*, with different types of payloads, over arbitrary shared dimensions (it is important to avoid pre-aggregations on predefined variables [48]);

v) a single perspective can help in determining a more general (not influenced by *observations* data types) set of guidelines to deal with the trade-off between collecting too many data and not exposing enough information on system behaviour.

Because of this, we believe the proposed model can provide a meaningful answer to **Q1**.

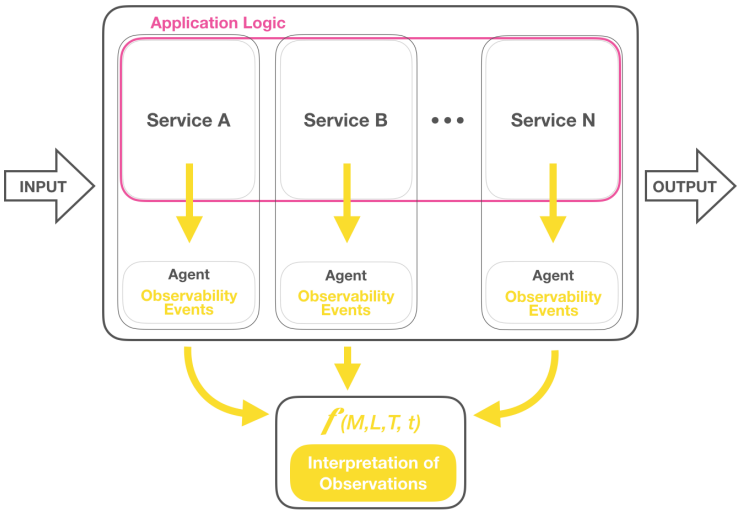

Fig. 4.2 Interpretation of observations.

Nevertheless, we claim metrics, logs and trace data remains the three main abstractions to reason on how to expose and make sense of system behaviour. The proposed model makes easier to process them also together and allows to process same data interpreting it under multiple abstractions, but they define different payloads, and each of them has its specificity. This consideration must be taken into account while instrumenting the system and in determining processing mechanisms for fine-grained access. For these reasons, the *interpretation of observations*, defined in Section 3.2.2, is still meaningful in approaching **OP2** (Figure 4.2) and this second problem can be decoupled from **OP1**.

4.2 Processing observations

As described in Chapter 3, processing of *observations* requires an engine able to process data flowing from the different components of the software system and to timely manage complex rules to make sense at runtime of system behaviour. Given the three requirements identified for **Q2**, we claim that the **OP2** belongs to the domain of *Information Flow Processing* [17].

In particular, the requirement **Q2.*R2*** call for a stream-based approach to achieve low latencies and to process data as soon as they arrive from *sources*. Storing operations and static analysis, if needed, are demanded to *information sinks*. The other two requirements, instead, are related to *processing rules* and must be taken into account to select a stream processing engine capable of handling *observations* as described by **Q2.*R1*** and **Q2.*R3***.

Information Flow Processing engines can be described through two models: (i) the *data stream processing* model, processing streams as inputs to output new data streams (traditional data processing applied to streams, e.g., DSMS[4]) and, (ii) the *complex event processing* (CEP) model, consuming stream data as notifications of events and processing them to produce higher-level events from particular detected patterns.

As pointed out in **Q2.*R1***, to effectively process *observations* it's desirable to have the possibility to express both *transforming* and *detecting* rules. Originally, this kind of rules were respectively implemented by DSMS (*transforming*) and CEP systems (*detecting*), but currently there exist stream processing engines allowing to express both types of rules (e.g. Esper[5], Drools Fusion[6], Siddhi [61]). Also, with respect to **Q2.*R3***, it is possible to found engines adopting languages that provides enough expressiveness to *filter*, *aggregate* and *join* data despite of their schema and allowing for flexible data models.

Therefore, we would like to validate our claim showing that is possible to process *observations* effectively through a stream processing engine. To show the benefits of this approach also in existing production environments and without requiring a change in instrumentation, we do not choose to adopt the model proposed in Section 4.1 and we decided to opt for an already widespread data model.

Stream processing solutions for logs are quite diffused (e.g., Kafka-based pipelines [40], *Humio*[7] or *Honeycomb*[8]) and also stream-oriented pipelines for metrics have been investigated (e.g., research works [8, 58], *Tick* stack by *InfluxData*[9]).

[4] Data Stream Management System
[5] Esper http://www.espertech.com/
[6] Drools https://www.drools.org
[7] Humio https://www.humio.com
[8] Honeycomb https://www.honeycomb.io
[9] InfluxData https://www.influxdata.com/time-series-platform/

State-of-the-art open-source distributed tracing tools (e.g. Jaeger [56] and Zipkin[10]) are, instead, still relatively immature with regard to stream processing of trace data. They 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. These tools enable a static analysis of trace data and currently do not implement a solution for dynamic analysis. However, as pointed out in the literature on distributed tracing, dynamic analysis plays an important role [47, 54] and can be effectively paired with the static one to supervise the system behaviour in large-scale systems [38].

Given **OP1** and considering research works and technical reports cited, we would like then to show benefits of near real-time processing also with respect to trace data. Therefore, we elicit a specification for a set of tools, we named **Trace Stream Processor** (TSP), that can implement this approach and can be used to evaluate it. We focus then on request-scoped *observations* highlighted in Figure 4.3, and in particular we consider the OpenTracing specification, offering a widely accepted standard for trace data. It should be pointed out that we also consider overlapping zones (shown in Figure 4.3) since this specification can easily embrace all kind of request-scoped *observations*. Indeed, *span logs* are request-scoped records, and metric, e.g. related to resource usages of a specific request, can be reported as a *span tag*.

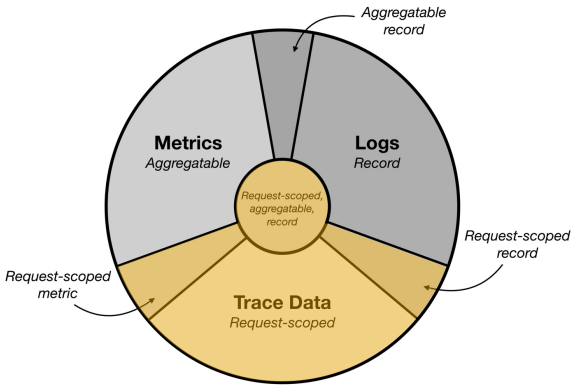

Fig. 4.3 Request-scoped observations.

On these assumptions, to validate our claim, we designed *Kaiju*, a TSP prototype implementing a stream-based approach for request-scoped *observations*, and *Rim*, an environment to reproduce the target context (as depicted in Figure 4.4). In this way, following the *Design Science* framework discussed in Section 2.5, we were able to design *Kaiju* exploiting an iterative process of evaluation of the artifact in its context. Require-

[10]Zipkin https://zipkin.io

ments for both the artifact and the context have been iteratively elicited focusing on their interactions but to facilitate the discussion we report them separately.

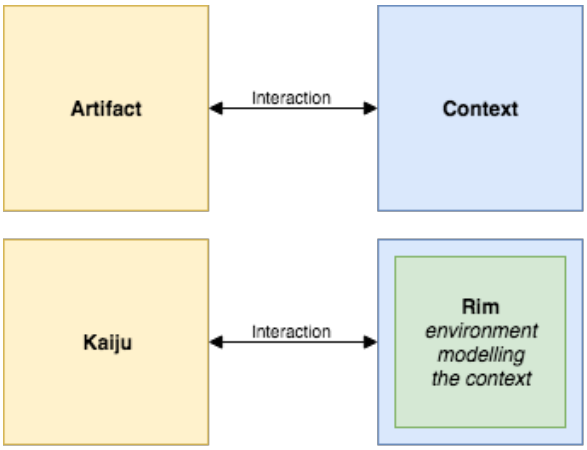

Fig. 4.4 *Kaiju* artifact interacting with *Rim*.

4.2.1 Rim

In this section, we describe the specification for *Rim*, an environment that should be representative of the context addressed by *Kaiju*. The context is related to the problem of *observability* for software, as described in Chapter 3, and in particular to the processing of request-scoped *observations* produced by running distributed software. Since we do not have a real production cluster to monitor nor a dump of these types of data, we decided to build a configurable and reproducible environment that can be used to introduce arbitrary issues and to produce relevant data to evaluate our prototype.

Rim should represent a running distributed application in a usual production scenario. Therefore, the application should exploit a service-oriented architecture decomposing the business logic in multiple micro-services. To produce request-scoped *observations* the components should be instrumented with OpenTracing API to propagate metadata and produce data on incoming requests. Moreover, we would like to observe all traces produced, i.e., it should be possible not to implement sampling strategies. To be run in a distributed fashion it should enable the deployment of services on different machines and, to guarantee portability, we would like to made *Rim* micro-services shippable as containers. To ensure the application can scale, it should also provide an easy method to deploy multiple instances. In this way, it is possible to cope with a high number of requests and, consequently, to tune the amount of trace data produced. Furthermore, we want to instrument *Rim* to output also other types of observations (e.g. not request-scoped

logs and metrics) to compare the different outputs and to enable approaches taking into account different types of data.

Given these specifications, the core feature of *Rim* should be the possibility to tune some parameters, when launching the application, in order to introduce or remove programmatically a set of issues on latencies and concurrency of operations. To conclude, *Rim* should allow generating load tests to automate requests and the production of trace data. Moreover, since instances can also be configured in different ways, it should also enable the possibility to select how the traffic is shared among them.

In summation, we elicited the following specification for *Rim*:

Rim.*1* The application should be composed by multiple micro-services.

Rim.*2* Each service should be instrumented with the OpenTracing API and no sampling strategies implemented.

Rim.*3* Each service should be isolated and deployable independently from other services.

Rim.*4* Possibility to run multiple instances of the application.

Rim.*5* Generate also other types of observations (other than trace data).

Rim.*6* Possibility to increase-decrease latencies/concurrency of operations.

Rim.*7* Possibility to generate systematic and reproducible load tests to target instances.

Given the specification provided, the *Rim* environment allows: (i) to easily create a stream of request-scoped *observations*, (ii) to emulate a production scenario with multiple instances of a multi-node application, and (iii) to tune the volume of data produced and the set of issues introduced in the system.

In this way, we created a white-box system[11] to check the effectiveness of tools and methodologies to observe the system status at runtime.

4.2.2 Kaiju

The main specifications for *Kaiju* are related to the requirements identified for **Q2**:

TSP.*1* Enable efficient querying on trace data for (near) real-time fine-grained analysis (**Q2.*R2***, **Q2.*R3***) coping with delayed, missing and out-of-order data.

[11] *Rim* is an all respects a distributed application, therefore when it is deployed it can be affected to common unpredictable issues of distributed systems (e.g., network partitioning).

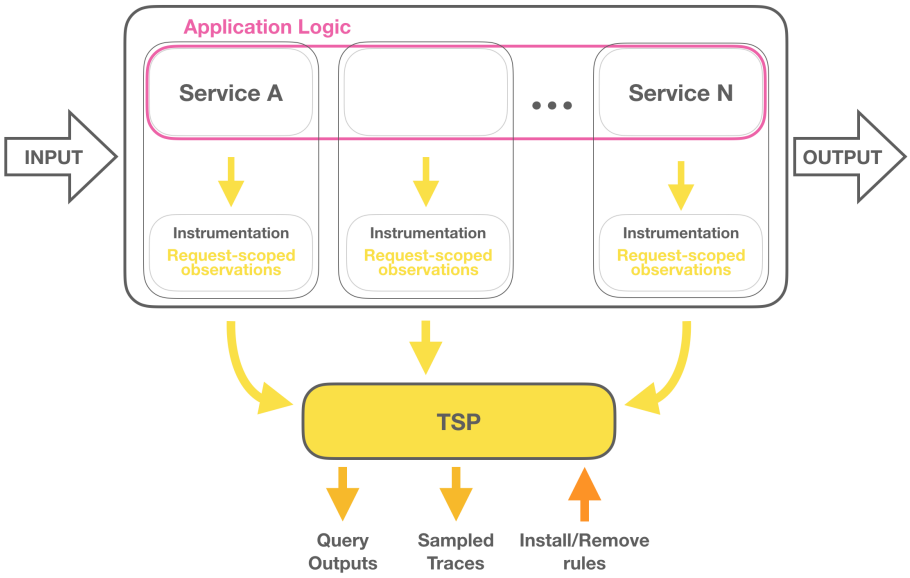

Fig. 4.5 A Trace Stream Processor (TSP) as described by the specification.

TSP.*2* Enable analytical continuous querying on trace data, also exploiting causality relationships within the same trace (**Q2.*R1***[a)], **Q2.*R3***).

TSP.*3* Enable the possibility to emit events and alerts about incoming trace data and to detect complex patterns. (**Q2.*R1***[b)], **Q2.*R3***).

Further specifications are related to *rules management* and to *sources* and *sinks* of the stream processing engine:

TSP.*4* Enable ingestion of trace data from instrumented software components.

TSP.*5* Enable possibility to install and remove rules at runtime.

TSP.*6* Enable possibility to output or store query results.

While metrics and logs have not a widely accepted and structured standard, the OpenTracing initiative offers a recognised and diffused vendor-agnostic specification for trace data, and so we decided to focus on this standard for the *Kaiju* prototype, detailing specification **TSP.*4*** as follows:

TSP.*4* Enable ingestion of trace data from components instrumented with the OpenTracing API.

An additional specification is related to the high volume of trace data generated by incoming requests and often requiring sampling. Large complex systems can process

millions of requests per second, so tracing systems, to avoid high overhead and to reduce the storage required, usually apply sampling policies selecting traces to-be-collected (constant or probabilistic, system- or service-wise, static or adaptive with respect to the actual workload). The main problem is related to the ingesting delay due to indexing-based stores currently used in distributed tracing tools' pipelines. Through a stream processor we can apply tail-sampling (a posteriori sampling strategies), i.e., we can elaborate all data storing only the relevant ones and avoiding the overhead of storing and then discarding them[12].

TSP.*7* Enable a posteriori sampling strategies to store relevant traces for static analysis.

In Figure 4.5 we highlight some of the specifications reported showing required interactions for a TSP.

To conclude, we elicit a non-functional specification related to resources consumption. Usually, tools for *observability*, share resources with applications supervised, so we want to guarantee a TSP can be deployed even if few resources are available to observe the system:

TSP.*8* Lightweight component requiring few resources for basic deployments.

4.3 Crossing the streams

The *observability event* model, proposed in Section 4.1.2, mainly addresses a problem of *variety* between *observations* data-types. It considers time as a first-class citizen (**Q1.*R1***), it guarantees functional interoperability (**Q1.*R2***) and the possibility of defining a shared terminology (**Q1.*R3***). On the other hand, the processing model proposed in Section 4.2 allows to process temporal relations (**Q2.*R1***) and handle complexity (**Q2.*R3***) through expressiveness of the processing language, but it mainly addresses a problem of *velocity* in processing *observations* (**Q2.*R2***).

Ideally, we would like to combine our two contributions to process different types of *observations* simultaneously. However, *Kaiju* is not suitable to handle data heterogeneity because, as a stream processor, it does not support semantic interoperability [23]. Indeed, *Kaiju* needs a set of adapters for each type of *observations* we want to ingest. In Section 5.3, we describe an extension of *Kaiju* that evaluates this possibility.

On the other hand, the challenge of taming *velocity* and *variety* simultaneously was addressed by the *stream reasoning* research area as RDF Stream Processing (RSP). RSP

[12] Networking overhead in data transfer should be anyway considered.

combines concepts from knowledge representation and reasoning with a stream-based approach [23]. Moreover, RSP exploits semantic web technologies (see Section 2.4.1), in particular the RDF data model and SPARQL, to guarantee **functional** and **semantic** interoperability. Therefore, we envision an RSP approach that fulfils requirements identified for **Q1** and **Q2** and requires, in addition, to:

RSP.*1* Model *observations* using semantic technologies

RSP.*2* Enable an RDF serialisation of *observations*.

RSP.*3* Provision queries in an RSP dialect (e.g., C-SPARQL, CQELS-QL and SPARQL$_{stream}$).

In the literature, it is possible to find research works modelling ontologies for logs and working on them as RDF graphs ([26, 52] and RLOG[13]). Moreover, a related work on *stream reasoning* applied to metrics analysis exists [50]. Even if it does not define an ontology, given the currently ongoing *OpenMetric* effort towards standardisation for the metric format, we decided to not focus on this.

In this work, we choose to focus on trace data exploiting the work done with *Kaiju*, and the experience made in processing this type of data. Therefore, to fulfil the specification above and evaluate this approach: (i) we design an ontology for trace data based on the *OpenTracing* specification [35], (ii) we generate a stream of trace data annotated in RDF, (iii) we consider an RSP engine consuming trace data as RDF graphs, and we investigate a set of queries in the considered RSP dialect.

[13] RLOG - an RDF Logging Ontology http://persistence.uni-leipzig.org/nlp2rdf/ontologies/rlog/rlog.html#

Chapter 5

Implementation

In this chapter, we describe implementations of prototypes modelled and specified in Chapter 4. In Section 5.1, we explain the implementation details of *Rim*, and in Section 5.2 the ones of *Kaiju*, fulfilling specifications for a Trace Stream Processor (TSP). In Section 5.3, we describe how we extend *Kaiju* to process also other types of *observations*. In Section 5.4, we detail the integration of *Kaiju* with an RSP engine.

5.1 Rim

In this section, we describe the implementation of *Rim*, i.e., an environment emulating a running distributed application, instrumented through the OpenTracing API and configurable to introduce issues in its behaviour at runtime. We follow the specification discussed in Section 4.2.1 explaining how *Rim* fulfils it.

Rim is based on HotR.O.D.[1], a demo application emulating a "ride sharing" application and instrumented to send trace data to the *Jaeger* distributed tracing tool. HotR.O.D. is a GoLang[2] application composed of four micro-services plus two emulated data storage, connected as depicted in the component diagram in Figure 5.1.

This basic application offers the user the possibility to make only one type of request interacting with the *Frontend* service. Each request made crosses all four services interacting as shown in the sequence diagram of Figure 5.2:

(i) *Frontend* service provides a simple web-app to make HTTP GET requests to the `\dispatch` endpoint. Clicking one of the four buttons available, each related to a specific `customerId`, it is possible to ask for a driver to be dispatched at the

[1] HotR.O.D. - Rides on Demand https://github.com/jaegertracing/jaeger/tree/master/examples/hotrod
[2] GoLang https://golang.org

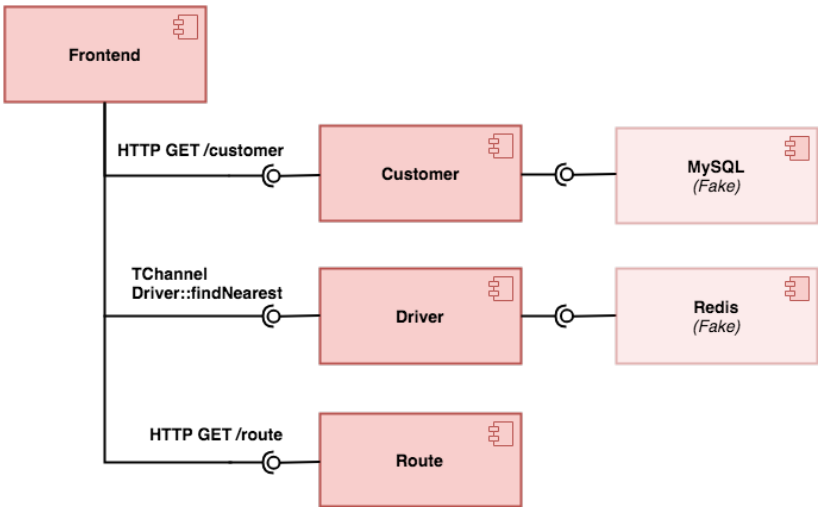

Fig. 5.1 HotR.O.D. component diagram. The HotR.O.D. application is composed by four microservices (*Frontend*, *Customer*, *Driver* and *Route*) plus two emulated data storage.

customer location. The request is sent to the back-end service that calls all others micro-services and responds with the driver's license plate number and the expected time of arrival.

(ii) *Customer* service is called through an `HTTP GET` call and it emulates a query to an SQL database returning data about the customer identified by the given `customerId`: `location` and `customerName`.

(iii) *Driver* service exposes the `findNearest` service through Thrift[3] over TChannel[4]. Emulating a Redis database, it randomly generates ten drivers available in the area of the customer and for each of them returns the `driverId` (license plate number) and `locationDriver`.

(iv) *Route* service is called through an `HTTP GET` call and computes the ETA[5] from the driver to the customer location.

The four microservices are instrumented with the OpenTracing API for GoLang[6] and the *Jaeger* tracer is configured to handle traces (sampling is not active). In order to exemplify the goal of distributed tracing some operations are intentionally misconfigured or not optimised (highlighted in yellow in Figure 5.2): latencies of some operations are

[3]Thrift https://thrift.apache.org
[4]TChannel https://github.com/uber/tchannel
[5]Estimated Time of Arrival
[6]OpenTracing-Go https://github.com/opentracing/opentracing-go

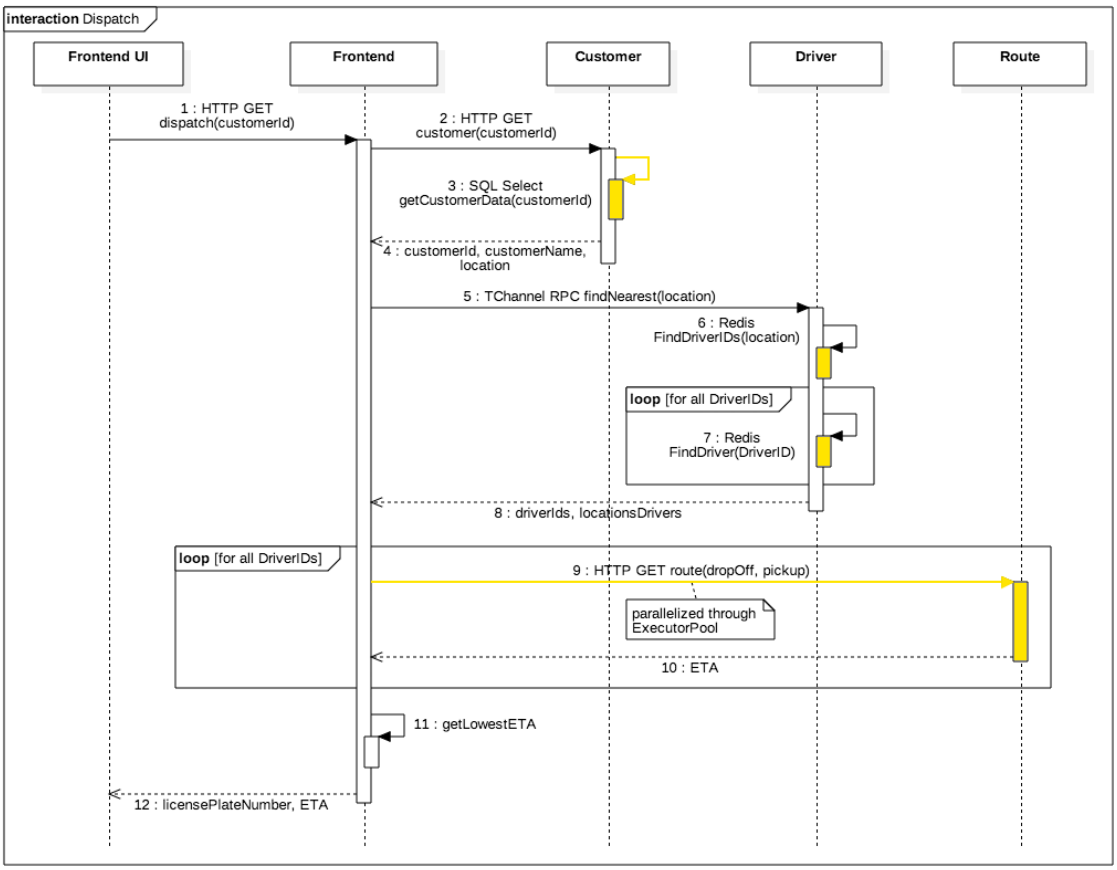

Fig. 5.2 Sequence Diagram of HotR.O.D. requests.

The diagram represents requests made through the UI or directly to the `/dispatch` endpoint of service *Frontend*: (1) HTTP GET request asking for a driver to be dispatched at the customer location identified by the `customerId`. (2) *Customer* service is called through an `HTTP GET` call to retrieve customer data. It emulates a query to an SQL database (3) returning data about the customer (4). (5) *Driver* service is then called through TChannel. It emulates a Redis query to find drivers available near customer location (6) and for each of them makes another query (7) returning the `driverId` and `locationDriver` (8). (9) *Route* service is called for each driver returned to compute the ETA from the driver to the customer location (10). (11) *Frontend* service select the lowest ETA and returns it with the correspondent driver's license plate number, i.e., the `driverId` (12). In yellow are highlighted operations intentionally not optimized in HotR.O.D. for demo purposes.

manually increased, some errors are randomly generated, and artificial bottlenecks are created.

As it is HotR.O.D. satisfies only partially our specification for an experimentation environment (**Rim**.*1* and **Rim**.*2*). In order to guarantee service isolation (**Rim**.*3*), each service should be deployed separately on a different process/machine. Therefore, we changed the

networking configurations in HotR.O.D. to enable each service to run as a different process on a different machine and thus to enable containerization of services. The original repository provides a Docker image allowing to run all four services or separated services but only if they are deployed in the same machine (communicating on `localhost` ports). We build a unique Docker[7] image for HotR.O.D.[8] that can be used to run separately each service and a `docker-compose`[9] file to instantiate HotR.O.D. with each service running on a different container[10]. To allow spawning more than one instance (**Rim**.*4*) we made the `docker-compose` file configurable through environment variables. In this way, as shown in Listing 5.1 it is possible to easily launch multiple instances of the application also on the same machine, reachable at different addresses and not clashing exploiting project namespace (`-p` option of `docker-compose`). To also generate other types of *observations* (**Rim**.*5*) and obtain a request-scoped metric we add a tag to the *Route* service span to report the CPU used to calculate the ETA. Moreover, HotR.O.D. already generates request-scoped logs within spans and the service *Route* already exposes at the endpoint `/debug/vars` a bunch of metrics in the Prometheus format.

In order to enable scalability, we guarantee a "for-instance" scaling, i.e., to scale the application we can run another instance of all four services. Each instance is spawned and configured through the `docker-compose` file and has its internal sub-network exposing only the *frontend* endpoint. We opted for this solution because it ensures requests sent to a specific *frontend* service are served only from micro-services of that instance, giving guarantees on the configurations provided for the given instance. To clarify this concept we show an example, let's suppose we want to launch two instances configured differently, we can execute the following commands:

```
1 HOTROD_INSTANCE="hotrod1" FIX_DB_QUERY_DELAY="--fix-db-query-delay=2ms" HOST_PORT_FRONTEND=8080
      docker-compose -f hotrod-docker-compose.yml -p hotrod1 up
2 HOTROD_INSTANCE="hotrod2" HOST_PORT_FRONTEND=8090 docker-compose -f hotrod-docker-compose.yml -p
      hotrod2 up
```

Listing 5.1 Examples of HotR.O.D. instantiation in *Rim*.

In this example, the instance `hotrod1` will have a simulated query delay in service *customer* slower than the default value (300 ms) of instance `hotrod2`. Since we do not allow to scale each service independently, we can ensure that each request sent to the endpoint `:8080/dispatch` will have the query delay configured for `hotrod1`, and each

[7] Docker https://docs.docker.com
[8] Image available on DockerHub https://hub.docker.com/r/marioscrock/rim/tags/
[9] Docker-compose https://docs.docker.com/compose/
[10] *Rim* https://github.com/marioscrock/Rim

SD	Description	Flag	Default value
3.	Latency SQL query getCustomerData	`-fix-db-query-delay`	*300ms*
3.	Mutex on DB access	`-fix-disable-db-conn-mutex`	*enabled*
6.	Latency Redis query FindDriverIDs	`-fix-redis-find-delay`	*20ms*
7.	Latency Redis query FindDriver	`-fix-redis-get-delay`	*10ms*
9.	Number of workers calling *Route* service	`-fix-route-worker-pool-size`	*3*
9.	Latency ETA calculation	`-fix-route-calc-delay`	*50ms*

Table 5.1 Configurable parameters in launching HotR.O.D. instances in *Rim*.

Configurable parameters in launching HotR.O.D. instances in *Rim*. *SD* indicates the number of the correspondent operation in the sequence diagram in Figure 5.2. To avoid all spans having too similar values, latencies are generated during execution considering a Gaussian distribution with the given value as mean and a standard deviation of $\frac{value}{4}$.

request sent to the endpoint `:8090/dispatch` will have the default query delay configured for `hotrod2`.

To fulfil **Rim**.*6* and increase-decrease latencies/concurrency of operations we add a set of flags that can be set when launching the instance through the command line, or through environment variables in the `docker-compose` file. In this way, we avoid the need for recompiling the application. Configurable operations are highlighted in yellow in the sequence diagram in Figure 5.2 and described in Table 5.1.

To complete *Rim* implementation we need a way to generate requests on instances reproducibly and systematically (**Rim**.*7*). To this scope we built *MakeRequests*, a component of *Rim* environment composed by:

- a Javascript library `makerequests.js`[11] to generate in-browser load tests given a set of parameters,
- a UI integrated into the HotR.O.D. application to use the library, and
- a GoLang script, configurable with downloadable files produced by the library, to execute load tests on target instances.

We decided to build a Javascript library so that it can be integrated with the UI of the *Frontend* service (see Figure 5.3). In this way, it is possible to avoid resorting to external tools for load testing. The UI provided allow to choose the clickable object to make requests on, or a *random* selection (for each request made, between all clickable elements specified). Also, it allows to set a textual *seed* to initiate the random pseudo-generator to guarantee reproducibility. The default method to generate requests is to specify a *number*

[11] Makerequests https://github.com/marioscrock/makerequests.js

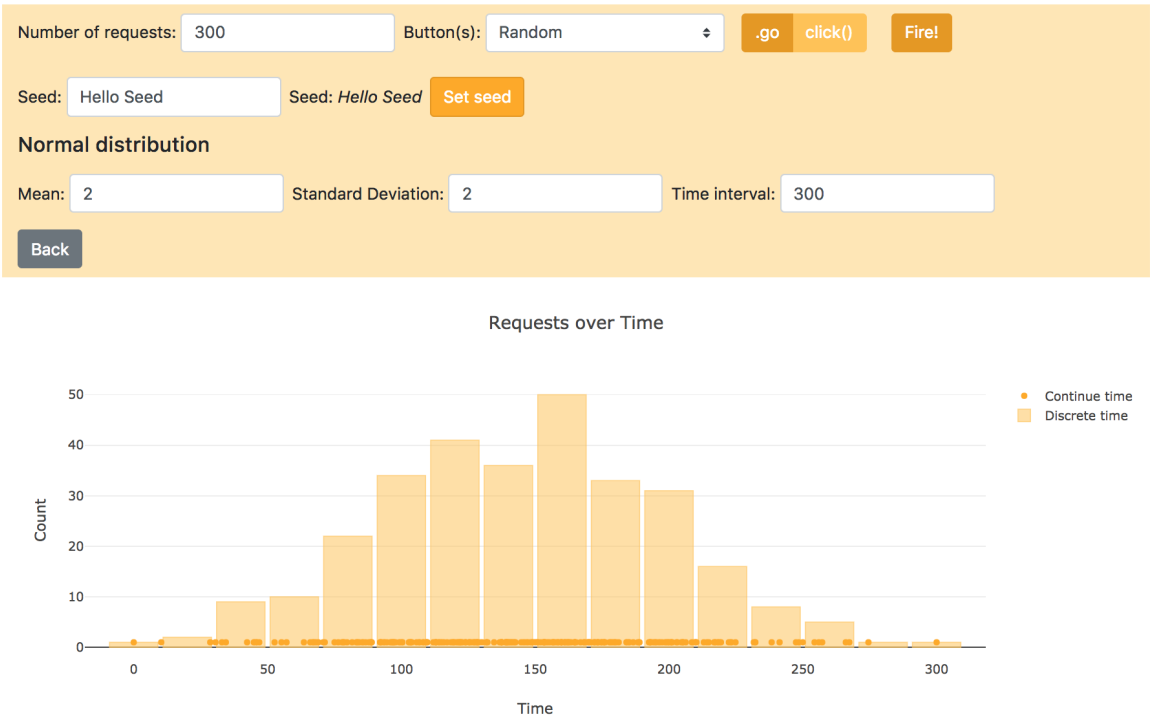

Fig. 5.3 Interface of the `makerequests.js` library.

of requests to be generated all at once. Otherwise, it is possible to select a distribution and provide parameters to determine how requests will be generated.

Modern browsers allow only for a limited set of concurrent requests, therefore, to generate a higher number of requests, we also implement an option to generate a GoLang file `makeRequestsTimes.go` containing endpoint URLs and timing of requests to be executed. This file can be run together with the `makeRequests.go` file provided to generate a set of go-routines executing requests as specified through the graphical interface. Moreover, the file can be easily tuned to implement more complex logic in the execution of the load test and to target multiple instances of the application.

5.2 Kaiju

In this section, we describe the implementation experience of *Kaiju*[12], our Trace Stream Processor (TSP) prototype, following the specification reported in Section 4.2.2. We discuss the choice of the stream processor engine, i.e., the core component of *Kaiju* responsible for the actual processing of trace data. We detail the mechanism to collect trace data,

[12] *Kaiju* https://github.com/marioscrock/Kaiju

and we explain how the design and implementation of the `kaiju-collector` fulfils the specification.

5.2.1 Stream Processor Engine

The first design choice we made is about the stream processing engine that should be used to process incoming request-scoped *observations*. We decided to use *Esper*[13], a state-of-the-art open-source complex event processing engine, and in this section we discuss how it fulfils specifications. Esper is a real-time streaming-capable engine[14] exploiting fast and optimised in-memory computing to process high volumes of data [15] (**TSP.*1***).

Esper queries are written using the Event Processing Language (EPL), a rich and flexible SQL-like query language that, combined with a powerful event representation mechanism, allows to write statements fulfilling specifications **TSP.*2*** and **TSP.*3***. EPL allows to express complex *transformation rules* using: (i) operators to select, aggregate, join, and filter, (ii) operators for data and time windowing, (iii) operators to define context partitioned queries, i.e., partition events on a defined context and execute queries on each partition, (iv) operators to define output-rate-limited queries, (v) operator to defined named windows and tables, and (vi) possibility to apply user-defined functions written in Java in statements and thus enabling whatever type of data processing. Moreover, a powerful *pattern*-matching mechanism allows to define also *detection rules*.

Combined to this high degree of expressiveness EPL also offers an *event representation language* that allows managing complex data structure, as trace data, and flexible schemas: (i) simple, indexed, mapped or nested properties, (ii) inheritance and polymorphism of event types, (iii) support to dynamic typing, (iv) create-schema syntax to define event types at runtime through statements, and (v) specific syntax for contained events.

Furthermore, Esper is lightweight in terms of memory, CPU, I/O, it has with no dependencies, and thus it can also be run as a single container with low overhead[16](**TSP.*8***).

[13]http://www.espertech.com/esper/

[14]From Esper documentation: *"Latency to the answer is usually below 10us with more than 99% predictability."*.

[15]From Esper documentation: *"It can process more than 6 million events per second per CPU."*.

[16]From Esper documentation: *"Minimum required Java version is fully supported. The compiler and runtime have no disk or other device or storage dependency and its memory and CPU use requirements depend only on what statements are needed."*.

5.2.2 Collecting trace data

To gather trace data from software components instrumented with the OpenTracing API we need a *Tracer* implementing the interface exposed from the API, i.e. a library to manage data produced[17].

Jaeger[18] is one of the *Tracer* implementations available, it is an open-source project and a state-of-the-art end-to-end distributed tracing system. Figure 5.4 shows the *Jaeger* architecture, discussed in details in Section 2.2.3. We choose to use it as a starting point to implement *Kaiju* collection mechanism for the following reasons:

(i) The `jaeger-agent` component is a sidecar process running on the application node and must be lightweight to do not add too much overhead. *Jaeger* components are written in GoLang, and this guarantees minimisation of overhead and optimisation of concurrency.

(ii) *Jaeger* decouples the language-specific tracer (`jaeger-client`) from the `jaeger-agent`, written in GoLang, forwarding asynchronously trace data to the backend. Indeed, the `jaeger-agent` component is the same despite the instrumentation library language, and it is the only responsible for communicating with the backend `jaeger-collector`. Therefore, modifying only this last component we can redirect the flow of trace data for all the `jaeger-client` libraries.

These are also the two main reasons why we choose to select *Jaeger* instead of *Zipkin*, an open-source alternative (Table 2.1 provides a detailed comparison).

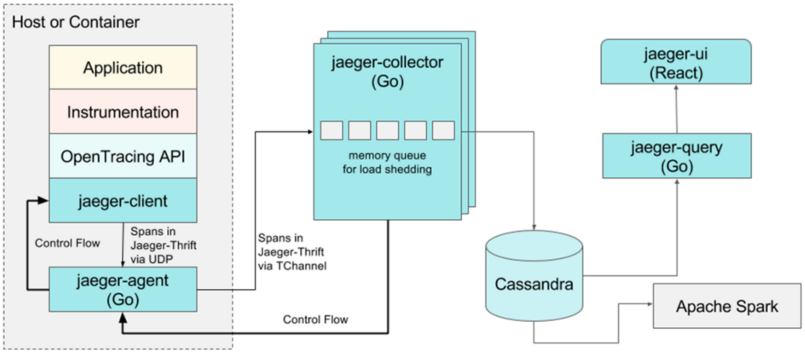

Fig. 5.4 The *Jaeger* architecture. Figure taken from [56].

In order to make *Kaiju* compatible with the OpenTracing ingestion format (**TSP.*4***) we made the design choice of assuming `jaeger-client` libraries as instrumentation

[17] Tracer documentation, https://opentracing.io/docs/overview/tracers/

[18] *Jaeger* https://github.com/jaegertracing. Since *Jaeger* is constantly updated we choose to select the stable release `1.5.0` (*Jaeger* version 1.5.0 https://github.com/jaegertracing/jaeger/releases/tag/v1.5.0).

libraries and to create a custom `jaeger-agent`. In this way, we can exploit a set of tracers available for all common programming languages and a lightweight sidecar component to asynchronously push trace data to the `kaiju-collector`.

To implement the proposed pipeline we need then to modify the `jaeger-agent` adding `kaiju-collector` as a registered collector implementing the *Reporter* interface. However, this poses another design choice between implementing `kaiju-collector` as the *MainReporter* or as an *AdditionalReporter* keeping `jaeger-collector` as the main one.

Since we are building a prototype, we choose the second option implementing the architecture described by the component diagram in Figure 5.5. Therefore, we deploy a *lambda architecture* [45], maintaining in parallel streaming and batch processing of the same data and allowing to: (i) use *Jaeger* storage and visualisation to analyse data also sent to *Kaiju*, (ii) to let `jaeger-collector` manage sampling policies (*control flow* in Figure 5.4) without imposing further requirements on `kaiju-collector`.

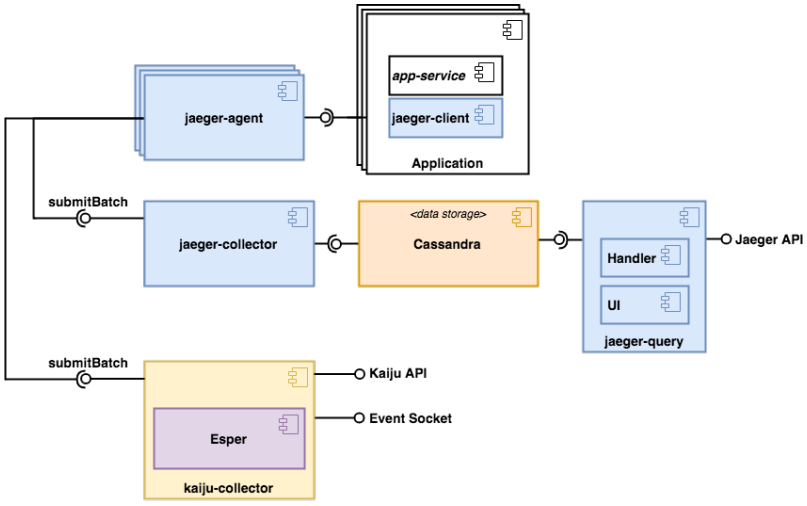

Fig. 5.5 Component diagram showing integration of *Kaiju* and *Jaeger*.

The last design choice to collect trace data in *Kaiju* is related to the network stack used to implement the *Reporter* interface. We decided to use *Thrift* over *TChannel* as done from *Jaeger* to report trace data to `jaeger-collector`. *TChannel* is a network multiplexing and framing protocol, it enables service discovery and fault-tolerance and it allows to develop RPC client and servers according to an interface definition in *Thrift*. *Jaeger* defines, using the *Thrift interface description language* (Thrift IDL)[19], the communication interface functions (reported in Listing 5.2) and the model, based on the OpenTracing specification, to represent and (de)serialize transmitted trace data (shown in Figure 5.6).

[19] Thrift IDL https://thrift.apache.org/docs/idl

The source code of `jaeger-agent` already contains a GoLang client for *TChannel* supporting the *Thrift* specification and it can be used also to implement a *KaijuReporter*. As a result, to enable dispatching of traces to `kaiju-collector` it is sufficient to:
(i) `jaeger-agent`: implement and register the *Reporter* interface for *Kaiju*, and
(ii) `kaiju-collector`: compile the *Jaeger*'s `.thrift` file and use *TChannel* to define a server implementing the specified interface.

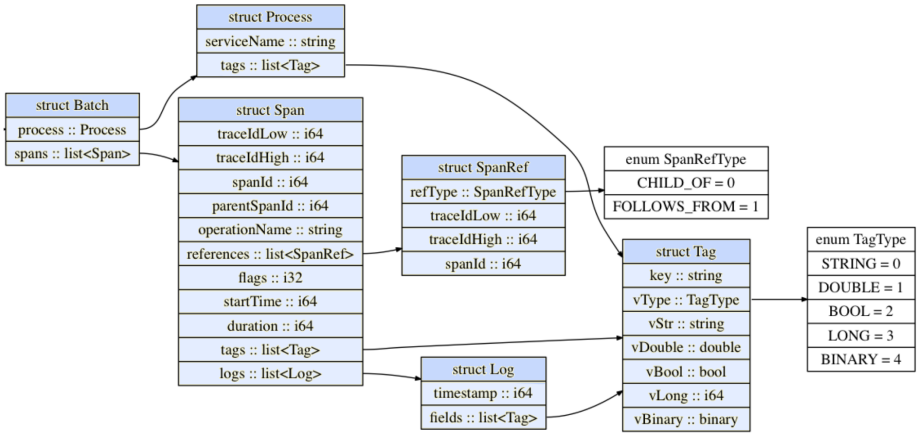

Fig. 5.6 *Jaeger* model described in Thrift.

The Thrift model implements the OpenTracing specification and its main concepts: *Span*, *Log*, *Tag* and *SpanRef*. Furthermore, it introduces the concepts of *Batch*, identifying a set of spans sent from one component, and *Process*, identifying a single component emitting spans. This model, here represented as a graph, is specified in the same file reported in Listing 5.2 in Thrift IDL[20].

```
1  # Copyright (c) 2016 Uber Technologies, Inc.
2  namespace java com.uber.jaeger.thriftjava
3
4  # BatchSubmitResponse is the response on submitting a batch.
5  struct BatchSubmitResponse {
6      1: required bool ok    # The Collector's client is expected to only log (or emit a counter) when
           not ok equals false
7  }
8
9  service Collector  {
10     list<BatchSubmitResponse> submitBatches(1: list<Batch> batches)
11 }
```

Listing 5.2 *Thrift* file defining the communication interface.

In Figure 5.7 we summarize the design choices made to collect trace data in *Kaiju*: (i) we choose to use the *Jaeger* tracer to support *Kaiju*'s collection mechanism, (ii) we

choose to define the `kaiu-collector` component as an *AdditionalReporter* implementing a lambda architecture, and (iii) we select Thrift over TChannel to transmit data.

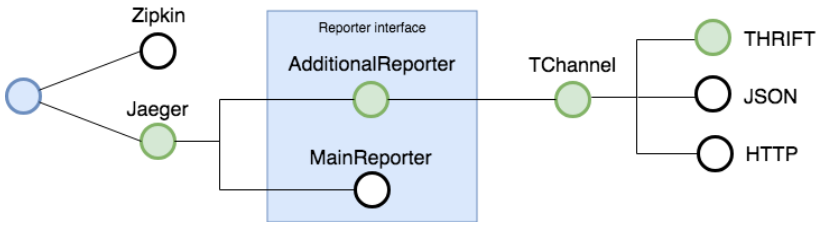

Fig. 5.7 Decision tree reporting design choices to collect trace data in *Kaiju*.

5.2.3 The `kaiju-collector` component

In this section, we will discuss design choices and implementation of the `kaiju-collector` component.

This component represents the core component of *Kaiju* and thus, we designed and implemented it to fulfil all its specifications, i.e., the ones identified for a TSP: (i) it is responsible of the actual processing of trace data exploiting *Esper* (**TSP.*1***, **TSP.*2***, **TSP.*3***, **TSP.*8***), (ii) it exposes the *Thrift*-defined interface as discussed in Section 5.2.2 to ingest trace data (**TSP.*4***), (iii) it exposes an API to enable interaction at runtime (**TSP.*5***), (iv) it allows to output query results through a set of implemented listeners components (**TSP.*6***), and (v) it allows to implement a simplified posteriori sampling strategies on file (**TSP.*7***).

Since Esper is a *Java*-based stream processing engine, consequently, we choose this language to develop the `kaiju-collector` component and easily integrate it.

The main class diagram in Figure 5.8 shows the overall class dependencies and packages structure. We will now detail the two packages separately.

The `collector` package

The `collector` package is composed by two main classes, the `Collector` and the `CollectorHandler` classes, as showed in Figure 5.8.

The `Collector` class contains the `main` method to run the component: (i) it opens the *TChannel* connection, (ii) it exposes the *Thrift*-defined interface `submitBatch` registering an instance of the *CollectorHandler* class to handle incoming trace data, (iii) it instantiates a static `ThreadPoolExecutor` serving a `BlockingQueue` to let other classes to schedule *Runnable* objects exploiting parallelism. To implement `kaiju-collector` as a server for the `submitBatch` interface, the *Thrift* IDL file has been compiled to *Java*

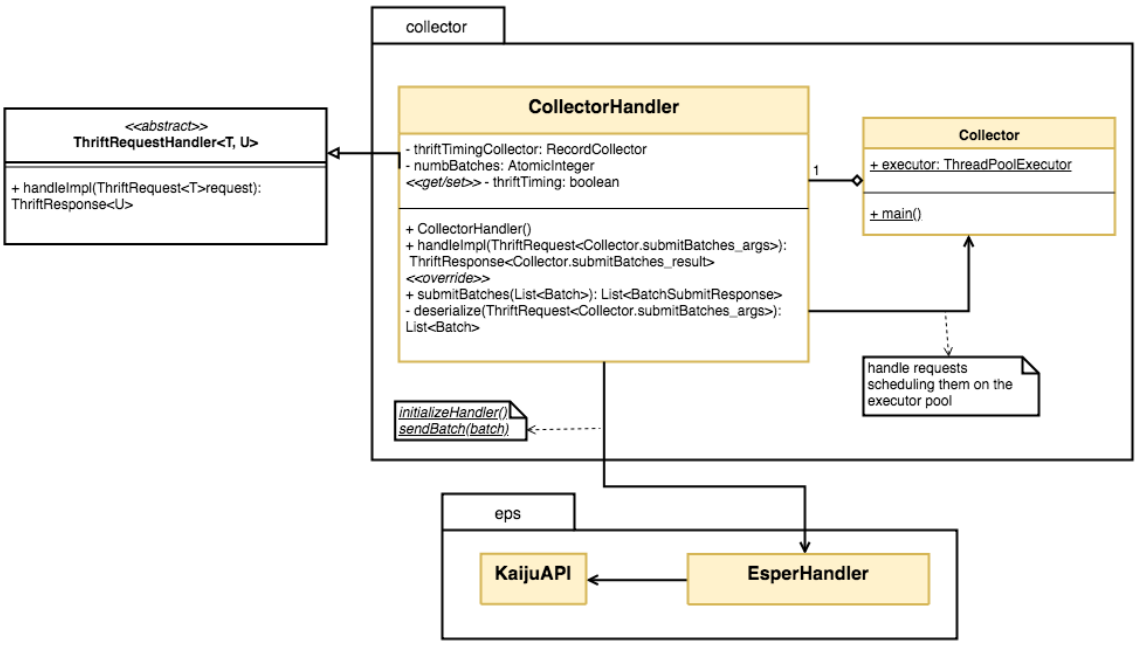

Fig. 5.8 *Kaiju* UML diagram: `collector` package and overall dependencies.

through the *Thrift* compiler and added to the project to manage dependencies of incoming *TChannel* requests. It is important to point out that, even if the interface exposed by `kaiju-collector` is the one used by *Jaeger* components, every *Tracer* could be modified to implement a client for the `submitBatch` interface and send data to *Kaiju*.

The `CollectorHandler` class, when instantiated, is responsible to initialize the *Esper* engine through the static method `initializeHandler()` of class `EsperHandler`. It also extends the `ThriftRequestHandler` class to implement the `submitBatch` method and manages the logic to handle incoming trace data. Each request is managed scheduling a *Runnable* object on the `Collector`'s executor: the request is deserialized through the *Thrift*-generated classes and the returned `Batch` object is sent to the *EsperHandler* class accessed statically. With respect to this last sentence it is important to highlight that *Esper* is guaranteed to be thread-safe and thus, multiple threads can push concurrently events in the engine. The `CollectorHandler` class also allows, setting the `thriftTiming` flag to `true`, to save to file records about time spent, for each `Batch`, to deserialize it.

The `eps` package

The `eps` package is composed by two main classes: the `EsperHandler` class handling the *Esper* engine and the `KaijuAPI` class handling the API exposed by *Kaiju*.

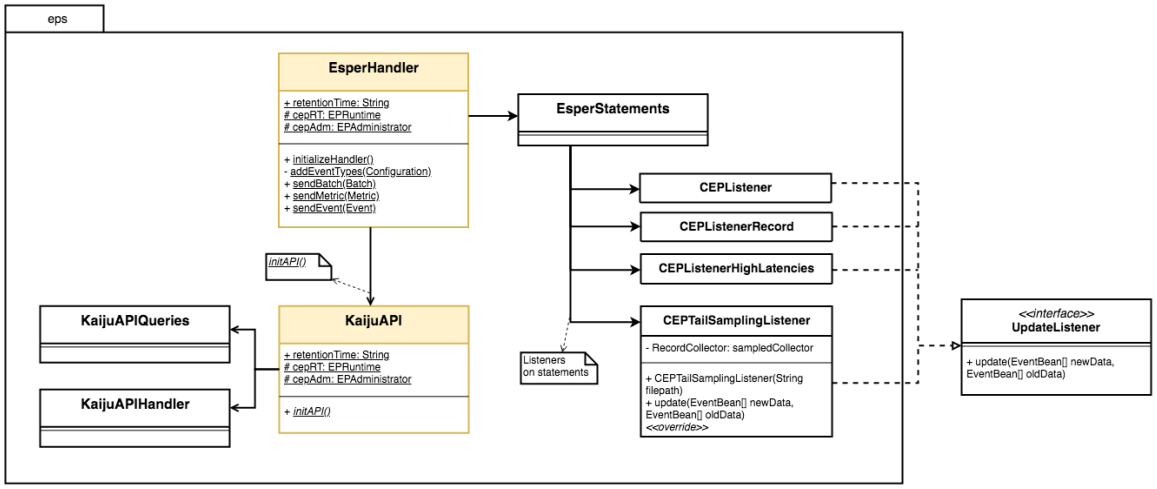

Fig. 5.9 *Kaiju* UML diagram: `eps` package.

The `EsperHandler` class can be accessed only statically and offers public methods to initialize the engine (`initializeHandler()`) and send to the engine incoming events. Since *Esper* allow to set Java POJO as event types, we opted to set all classes defined in the *Jaeger* model (5.6) as event types: *Batch*, *Span*, *Process*, *Log*, *Tag*, and *SpanRef*.

However, we need to consider the following issue, keeping all streams separated we lose relations between events. Taking, for example, a *Log* stream we would lose the information on the related *Span* unless we apply a traditional relational database approach exploiting additional data structures and then multiple joins on keys to reconstruct relations. Therefore, we decided to push incoming data on the *Batch* and *Span* streams and to exploit the contained-event syntax offered by *Esper* in statements to query contained events.

We keep the *Batch* and *Span* streams for two reasons: (i) even if also *Span*s are contained-events in *Batches*, we decided to build a separate stream since it is often more common to write statements on the *Span* stream rather than on the *Batch* one, and (ii) it is important to keep the *Batch* stream to avoid losing the relation between the *Span* and its *Process*.

Given these two considerations, to simplify queries, we also install a statement to build a *SpansWindow* named window making explicit the relationship between each *Span* and its *Process*[21].

[21] It is possible to point out that this is the reflection of an error in the *Jaeger*'s model: the relation between *Span* and *Process* is significant but exists only within the *Batch* object that is a transient concept only related to the reporting mechanism and not useful to analyse trace data. Indeed, the *Process* concept is not considered in the OpenTracing specification and data specifying it should be reported within the *Span Tag*s.

The `EsperHandler` class also accesses statically the `EsperStatements` class containing statements to be installed and launches the thread running the `KaijuAPI` through the `initAPI()` method.

The API is implemented through the *SparkJava* framework[22]. It fulfils the specification **TSP.*5***, but it can offer a wider interface as showed in Section 6.2. The two methods to install and remove statements at runtime exploit the interface provided by the `EPAdministrator` instance related to the initialized *Esper* engine. They define the following API:

- `POST /api/statement?statement=<stmt>&msg=<msg>`
  Installs the given statement (`<stmt>`) with a listener registered to output data to `kaiju-collector` logs with the specified message (`<msg>`). Returns the statement code.

- `POST /api/remove?statement=<stmt_code>`
  Removes the statement with the given code.

Note: All requests, to any API methods, return `status 400` and the related error message if *Kaiju* reports any error while managing the request.

The `eps.listener` package

To manage queries outputs and updates generated from statements, it is possible, for each *Esper* statement, to register instance of classes implementing the `UpdateListener` interface and defining the logic to handle events returned (an `EventBean` array). In our prototype, we build four different classes implementing this interface to show how *Kaiju* can fulfill **TSP.*6*** and **TSP.*7***.

- `CEPListener` class: A simple listener outputting data received as logs with a given message.

- `CEPListenerRecord` class: A simple listener outputting data received as rows in a specified file.

- `CEPListenerHighLatencies` class: A specific listener to handle spans reported because of anomalous latencies. We implemented this listener to show how it possible to define specific logic to handle specific statements. In this case, we report data in a custom defined `.csv` file to make late and further analysis on data reported.

[22] SparkJava http://sparkjava.com/

- `CEPTailSamplingListener` class: A specific listener saving spans reported to a `.json` file. We implemented this listener to provide an example of how to implement a posteriori sampling. However, other custom logic can be provided, e.g. writing spans to a database.

A posteriori sampling

To fulfill **TSP.7**, we discuss a possible method in *Kaiju* to implement an a posteriori sampling strategy:

1. Create a named window to retain the `traceId`s related to the traces to be sampled. We specify the `#unique` keyword to retain each `traceId` exactly once. The `<retentionTime>` is a place-holder and should be substituted with an appropriate interval.

```
1 create window TracesToBeSampledWindow#unique(traceId)#time(<retentionTime>) (traceId string)
```

Listing 5.3 Create named window to store `traceId` of traces to be sampled in EPL.

2. Exploiting the events hierarchy and implementing sampling strategies through a set of statements, it is possible to generate a stream of events inheriting from a *TraceAnomaly* event that identifies traces to be sampled and populates the named window.

```
1 on TraceAnomaly a
2 merge TracesToBeSampledWindow t
3 where a.traceId = t.traceId
4 when not matched
5   then insert into TracesToBeSampledWindow
6     select a.traceId as traceId
```

Listing 5.4 *On-merge-update* construct to sample traces reported as anomalous in EPL.

3. Exploiting the `rstream` of spans leaving a defined *SpansWindow* we can check if the span belongs to a trace to be sampled and report it to the `CEPTailSamplingListener`. This implicitly assumes that the span triggering the *TraceAnomaly* event arrives before the first span of the given trace leaves the *SpansWindow*.

```
1 select rstream * from SpansWindow as s
2 where exists (select * from TracesToBeSampledWindow
3 where traceId = (traceIdToHex(s.span.traceIdHigh, s.span.traceIdLow))
```

Listing 5.5 Select *Span*s to be sampled exploiting `rstream` in EPL.

5.3 Extending Kaiju

In this section, we describe how we extended *Kaiju* to enable ingestion of different types of *observations*. In particular, we add a further package, called `eventsocket` package, handling a web socket ingesting metrics and events in JSON.

The `eventsocket` package is constituted by: (i) the *EventSocket* runnable class handling a simple socket to receive events in `.json` and sending them to the *Esper* engine, (ii) the *ParserJson* class parsing valid `.json` strings, (iii) the classes implementing the events recognized by the parser (classes *Metric* and *Event*).

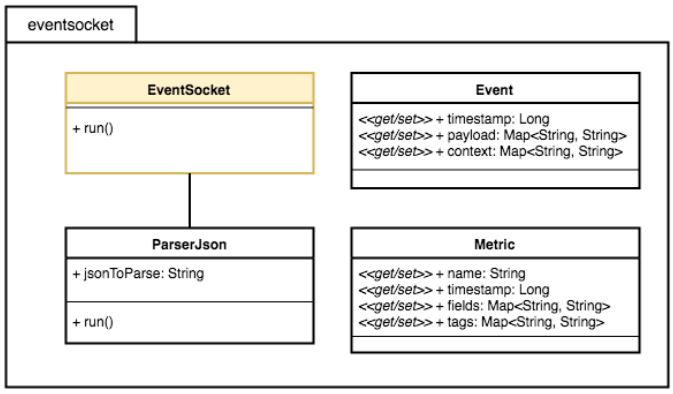

Fig. 5.10 *Kaiju* UML diagram: `eventsocket` package.

We choose to implement two recognised events type.

The *Metric* type represents the JSON metric format by InfluxDB[23]. We choose this format since to evaluate our prototype we exploit *Telegraf*[24], an agent to automatically collect/report metrics and supporting a wider set of formats and tools through an extensible set of adapters. Listing 5.6 shows an example of a metric sampled from a Docker container. Multiple *Metric*s can be reported together in a unique `.json` string through a JSON array with key `"metrics"`.

```
1  {
2    "fields":{
3      "usage_percent":1.700186933667084,
4      "usage_system":64963160000000,
5      "usage_total":133570111
6    },
7    "name":"docker_container_cpu",
8    "tags":{
9      "host":"09378849920c"
10   },
```

[23] https://github.com/influxdata/telegraf/blob/master/docs/DATA_FORMATS_OUTPUT.md
[24] https://github.com/influxdata/telegraf

```
11   "timestamp":1458229140
12 }
```

Listing 5.6 Example of *Metric* parsable by *Kaiju* in JSON.

The *Event* type, instead, is a simple JSON serialization fulfilling the data model proposed in Section 4.1.2 and representing a generic event capable of representing generic *observations*. An example reporting a new deployed commit is shown in Listing 5.7. Multiple *Event*s can be reported together in a unique `.json` string through a JSON array with key `"events"`.

```
1 {
2   "timestamp":1458229140,
3   "payload":{
4     "commit_msg":"Fix connection pool"
5   },
6   "context":{
7     "commit_id":"de9c1a087f47605cd7e33a585ee34d628a4a49b4"
8   }
9 }
```

Listing 5.7 Example of *Event* parsable by *Kaiju* in JSON.

5.4 Kaiju RDF

To implement a prototype of an RDF Stream Processor (RSP) consuming request-scoped *observations*, as discussed in Section 4.3, we decided to integrate an RSP engine with the *Kaiju* prototype.

In Section 5.4.1, to fulfill **RSP**.*1*, we detail an ontology for the OpenTracing specification [35] and an extension to this ontology to model *Jaeger* data model. In Section 5.4.2 to fulfill **RSP**.*2*, we explain how we generate a RDF stream of JSON-LD annotated trace data in *Kaiju*. Finally, in Section 5.4.3, to fulfill **RSP**.*3* we describe the RSP engine chosen and the adapter implemented to register the stream to the RSP engine and provision RSP-QL queries.

5.4.1 OpenTracing ontology

Given the OpenTracing specification [35], and following design criteria for ontologies [36], we design an ontological model for trace data.

First we identified the main concepts identifying *classes*:

(i) ***Trace*** identifies the trace related to a request

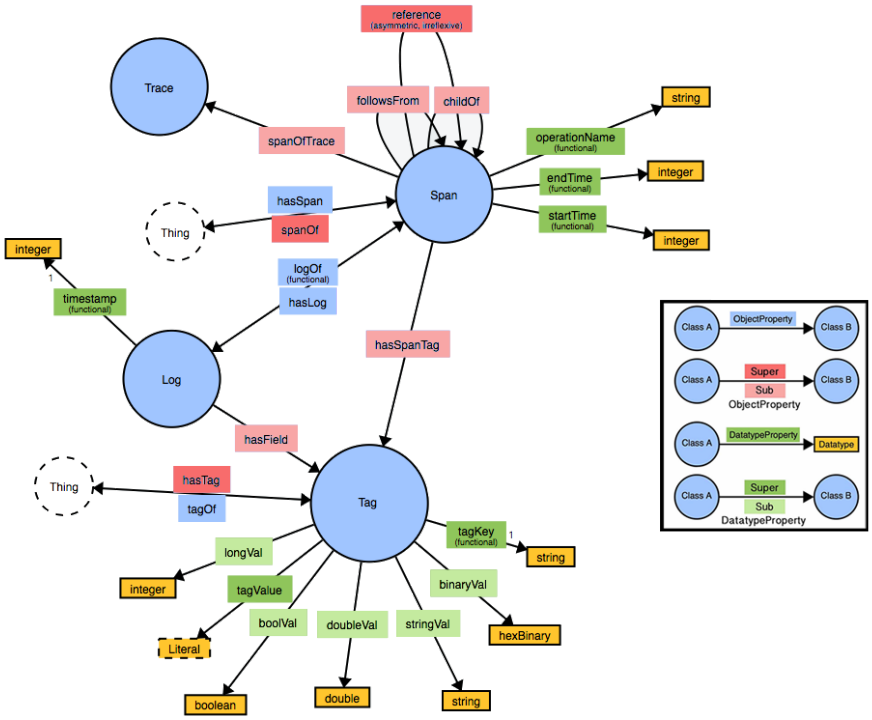

Fig. 5.11 OpenTracing ontology.

(ii) ***Span*** identifies units composing a trace

(iii) ***Log*** identifies timestamped key-value pairs related to a span

(iv) ***Tag*** identifies key-value pairs related to a span

Then we identified the main *object properties*:

(i) ***reference***, asymmetric and irreflexive property with *Span* as range and domain, with two sub-properties (*childOf* and *followsFrom*) identifying the possible relations between spans;

(ii) ***spanOf*** and the subproperty *SpanOfTrace* connecting a *Span* to the *Trace* it belongs;

(iii) ***hasLog*** connecting a *Span* to one of its *Log*;

(iv) ***hasTag*** and the two sub-properties *hasSpanTag*, connecting a *Span* to one of its *Tag*, and *hasField*, connecting a *Log* to one of its *Tag* (fields).

To conclude we identified the main *data properties*:

(i) ***operationName***: a `xsd`[25] : `string` to represent the operation performed by a *Span*;

(ii) ***startTime*** and ***endTime***: two `xsd:integer` to represent the starting and ending timestamp of a *Span*;

(iii) ***timestamp*** a `xsd:integer` to represent a timestamp (in the ontology used for *Log*);

(iv) ***tagKey*** a `xsd:string` to represent the key-value of a *Tag*;

(v) ***tagValue*** and all its sub-properties *longVal*, *boolVal*, *doubleVal*, *stringVal* and *binaryVal* to represent the possible values of a *Tag*.

We show the complete ontology in Figure 5.11.

Jaeger ontology

The OpenTracing ontology described do not covers entirely the conceptual data model used by *Jaeger*. Therefore, to represent the data model shown in Figure 5.6 and used from *Kaiju*, we extend the OpenTracing ontology to model:

(i) the ***Process*** class representing a process emitting spans,

(ii) the ***spanOfProcess*** object property, sub-property of the *spanOf* property, and relating each span to the process that emitted it,

(iii) the ***hasProcessTag*** object property, sub-property of the *hasTag* property, and relating each span to the set of tags describing it,

(iv) the ***serviceName*** data property to represent the name of the service run by the process,

(v) the ***traceId*** data property to represent the identifier of a trace,

(vi) the ***spanId*** and ***flags*** data properties to represent the identifier and additional metadata of a span, and

(vii) the ***duration*** data property for a *Span* substituting, without loss of generality, the concept of *endTime*.

In Figure 5.12 we highlight classes and properties introduced in the *Jaeger* ontology.

[25] XML Schema http://www.w3.org/2001/XMLSchema#

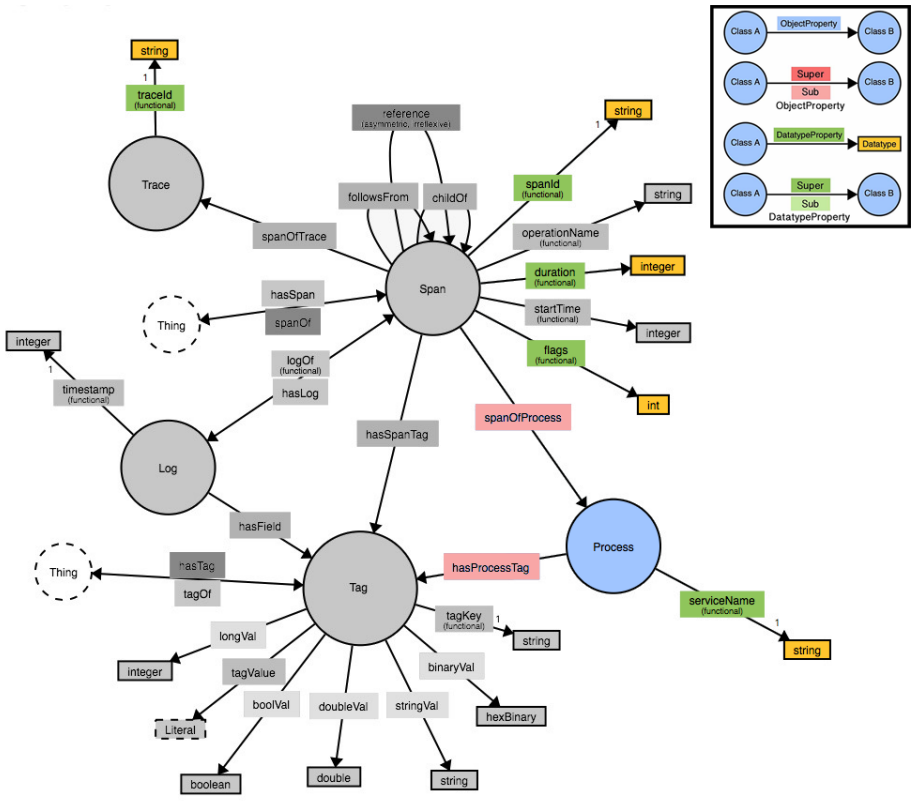

Fig. 5.12 *Jaeger* ontology extends the OpenTracing ontology.

5.4.2 Exposing an RDF stream in Kaiju

To feed the RSP engine with a stream of request-scoped *observations* we pose a further specification to *Kaiju*, i.e., it should expose incoming request-scoped *observations* in one of the RDF formats as a stream.

As shown in Figure 5.13, we decided to add a further interface in the `kaiju-collector` component that through a web socket[26] exposes data received as an RDF stream in the JSON-LD format.

Therefore, we extend the `kaiju-collector` component adding a further package (`websocket` package). We choose to use the JSON-LD serialisation format for the following reasons: (i) to easily represent the nested structures of the *Jaeger* data model (in JSON-LD lists are part of the data model whereas in RDF they are part of a vocabulary), and (ii) to provide a general interface that can also be processed as JSON and is not bound to the semantic web stack.

[26]Web Socket https://tools.ietf.org/html/rfc6455

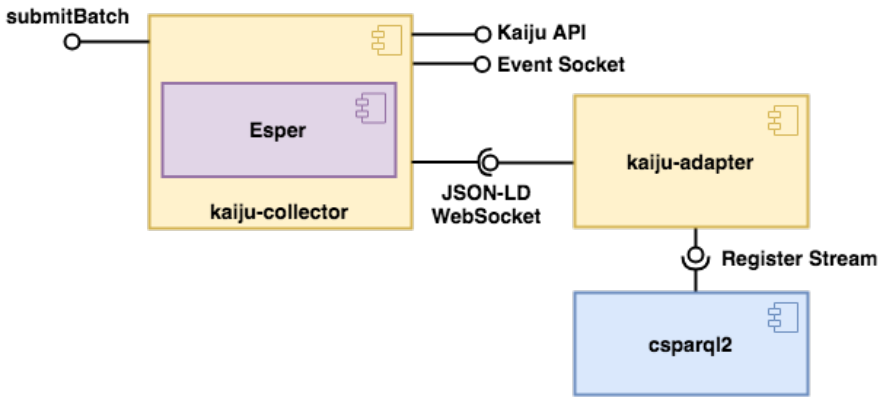

Fig. 5.13 Component diagram for *Kaiju* integrated with CSPARQL2.

In Figure 5.14 we show the complete class diagram for the `kaiju-collector` component. It represents the details of the `collector` package, updated to handle the web socket, and the overall dependencies among packages.

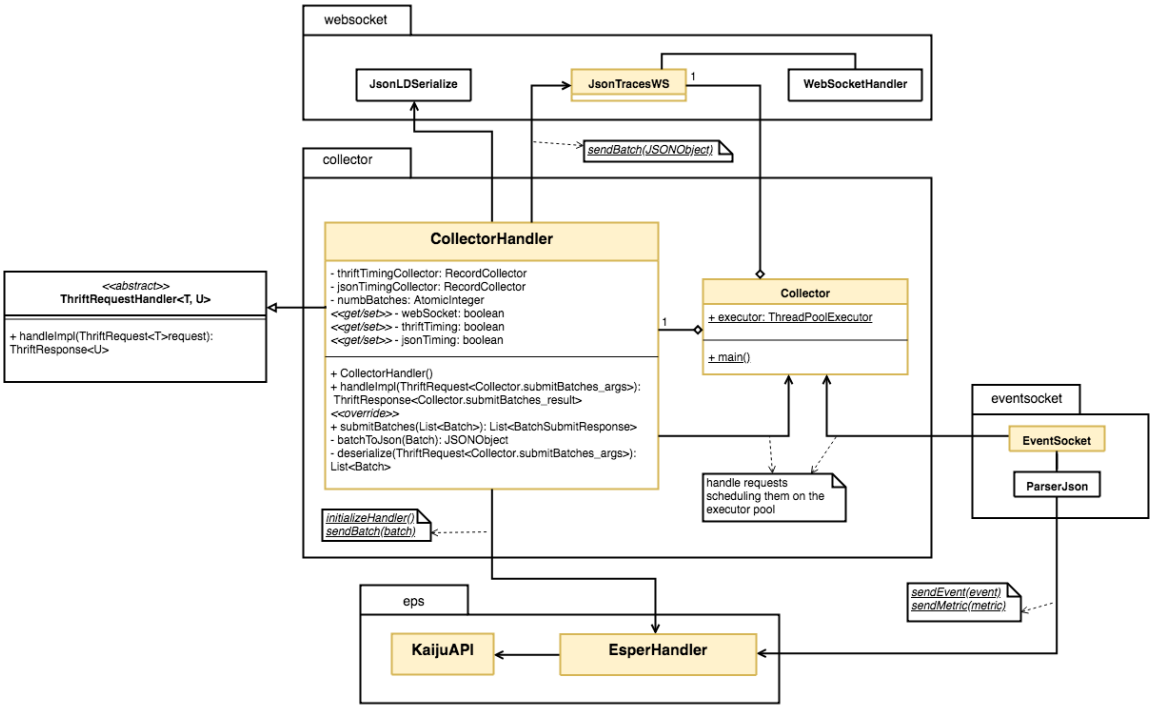

Fig. 5.14 *Kaiju* UML diagram: *collector* package and overall dependencies (with *websocket* package).

The `websocket` package

The `websocket` package is composed by three main classes as shown in Figure 5.15: the `JsonLDSerialize` class, the `JsonTracesWS` class and the `WebSocketHandler` class.

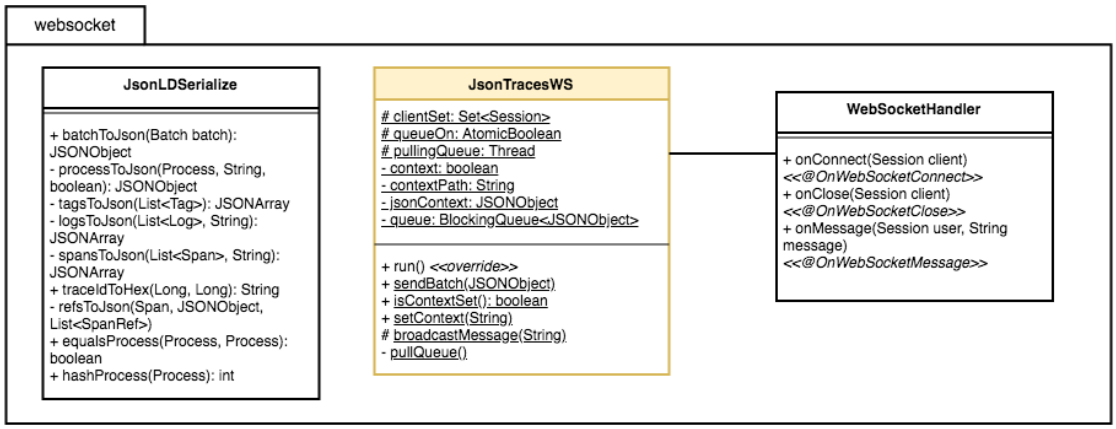

Fig. 5.15 *Kaiju* UML diagram: *websocket* package.

To expose data gathered from `kaiju-collector` as JSON-LD, we generated a JSON-LD context from the *Jaeger* ontology shown in Figure 5.12. Therefore, we created a `JSONLDSerialize` class to produce JSON strings from a *Batch* object that can be interpreted as an RDF graph through the JSON-LD context.

This poses a set of design choices to assign URIs[27] to data and to determine which data should be represented by the same entity in the RDF graph:

- a *Trace* is uniquely identified by its `traceId`,
- a *Span* is uniquely identified by its `traceId` and `spanId`,
- a *Process* is uniquely identified by its `serviceName` and the set of its tags (to this purpose we created a custom hash function generating the same hash for two processes if they have the same `serviceName` and the same tags despite of their order)
- a *Log* is uniquely identified by the `traceId` and `spanId` of the *Span* they are related to and the timestamp,
- a *Tag* is uniquely identified by its `key` and its `value`.

[27] Uniform Resource Identifier

The web socket is implemented through the *SparkJava* framework[28] and the *WebSocketHandler* class, an annotated class implementing the web socket API defined from the framework. The `JsonTracesWS` class is a runnable to launch the web socket, it offers static methods and fields for the *WebSocketHandler* and it allows through the `sendBatch(JSONObject batch)` static method to send a string serialization of the given `JSONObject` to each active session.

5.4.3 Adapter for CSPARQL2

In this section, we discuss the custom `kaiju-adapter` built to consume the stream exposed from *Kaiju* and redirecting it to the RSP engine. In this way, we can register the stream to the RSP engine and enable querying on it.

We choose to use the already implemented RSP engine CSPARQL2. CSPARQL2[29] is an RSP engine built through *Yasper*[30], a library to build RDF Stream Processing (RSP) engines according to the reference model RSP-QL [24]. CSPARQL2 exploits underneath *Esper* and *Jena*[31] to implement its functionalities and expose an interface based on *Jasper*[32]. Moreover, CSPARQL2 offers a query language fully compliant with the RSP-QL model.

The adapter built is composed by three main classes (as shown in Figure 5.16): the `CSPARQLKaiju` class, the `GraphStream` class and the `JsonLDSocket` class.

The `JsonLDSocket` class implements the *Jetty API* for a web socket client[33] and to be instantiated it requires a `RegisteredEPLStream` and a file-path to a JSON-LD context. It is implemented as follows: (i) it listens for incoming messages, (ii) it parses messages applying the given JSON-LD context through the Jena `RDFParser`, and (iii) it writes the parsed RDF graph on the given `RegisteredEPLStream`.

The `GraphStream` class extends the `RDFStream` class provided by *Yasper* and it abstracts the stream of data. Once instantiated with a given URI, it can be registered to an RSP engine through *Jasper*, and it can be set as *"writable"* passing the `RegisteredEPLStream` returned by Jasper. If a `RegisteredEPLStream` is provided, the `GraphStream` class can then be run to launch an instance of the `JsonLDSocket` class listening on the web socket exposed by *Kaiju* and writing data to the registered stream.

The `CSPARQLKaiju` class contains the `main` method and exploits an instance of the `Jasper` class providing an interface to configure the engine. In the `main` method, it

[28] SparkJava http://sparkjava.com/
[29] CSPARQL2 https://github.com/riccardotommasini/csparql2
[30] Yasper https://github.com/riccardotommasini/yasper
[31] Jena http://jena.apache.org
[32] Jasper https://github.com/riccardotommasini/jasper-ws
[33] Jetty API https://www.eclipse.org/jetty/documentation/9.4.x/websocket-jetty.html

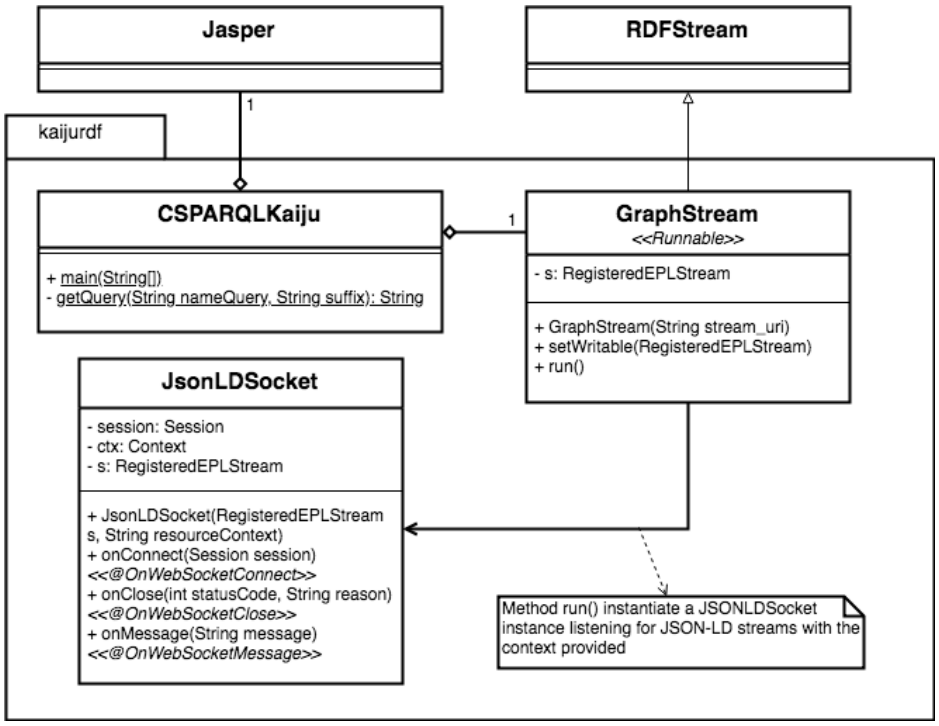

Fig. 5.16 UML diagram of the adapter for CSPARQL2.

instantiates the engine, it registers the `GraphStream` to consume *Kaiju* data as described, and it installs a set of provided RSP-QL queries.

Each query can be registered together with a `ResponseFormatter` handling its results: if it is a *SELECT* query the results can be logged or saved if it is a *CONSTRUCT* query the new stream can be in turn registered to the engine.

In this way, we implemented a prototype allowing to analyse potentialities of a stream reasoning approach applied to request-scoped *observations*.

Chapter 6

Evaluation

In this chapter, we present the procedures followed to evaluate the implemented prototypes and the results obtained. Since there isn't an established *benchmark* to evaluate processing of request-scoped *observations* and we would like to offer a comprehensive analysis of *Kaiju*'s capabilities we structured our evaluation as follows.

In Section 6.1 we discuss the deployment used for our evaluation. In Section 6.2 we apply *Kaiju* and *Jaeger* within *Rim* in the deployment described and we compare their behaviour. We point out differences and advantages of the two solutions, and we pose a baseline for *Kaiju* evaluation showing it can offer at least the same expressiveness of *Jaeger* in processing request-scoped *observations*. In Section 6.3 we consider the set of use cases identified for distributed tracing in [54]: we discuss how *Rim* can reproduce a set of issues related to the identified use cases, and to evaluate *Kaiju* we propose a set of rules to challenge them and we discuss their effectiveness. In Section 6.4 we show how, extending *Kaiju* to ingest other types of *observations*, we can enable pattern detection across different streams. Finally, in Section 6.5 we propose an explorative analysis on the application of stream reasoning to request-scoped *observations*, pointing out possible use cases and advantages.

6.1 Architecture and deployment

To evaluate our prototypes we decided to set up *Rim* in a cloud environment and to deploy both *Kaiju* and *Jaeger* to get the *observations*. Indeed, the modified version of the `jaeger-agent` described in Section 5.2.2 guarantees data produced by the *Rim* environment can be sent both to *Kaiju* and *Jaeger* instances. The simultaneous deployment has two goals: (i) we want to observe similarities and differences between the two approaches, (ii) we want to assess *Kaiju* correctness with respect to *Jaeger*.

As showed in Figure 6.1, the two tools share the collection mechanisms made by `jaeger-clients` and `jaeger-agents` and, to made a fair comparison with the single node `kaiju-collector`, we deployed only one instance of the `jaeger-collector`. Concerning *Rim*, since in real production environments, the agent collecting spans is thought to be deployed as a sidecar process of the component producing them, we build a Docker image[1] such that each HotR.O.D service is run together with the modified `jaeger-agent`. To complete the deployment we also add an instance of InfluxDB[2] and a set of Telegraf agents[3] to collect metrics about containers execution and to collect metrics exposed by *Jaeger* and *Rim* components. To feed the `kaiju-collector` event socket we exploit the same Telegraf agents filtering only metrics from *Rim* and we implement a simple socket client to send arbitrary *observations* in JSON.

We deploy software in containers using Docker[4], and we generate parametric docker-compose[5] files to easily ship prototypes in a cloud environment and configure their interactions and communications.

We utilise three EC2 instances on Amazon AWS[6] and we implement a VPN (virtual private network) to manage connections between different machines. We describe the three instances, their software requirements and their interactions and in Figure 6.1 we show the main component deployed on *M1* and *M2*.

M1: *Rim*

- *Instance type*: 1 Linux instance `m4.2xlarge` (8 vCPU | 32 GiB Mem)
- *Requirements*: Docker, docker-compose `v2`, GoLang `v1.10`, *Rim* Docker image (with `jaeger-agents` as sidecar process), *Rim*'s parametric docker-compose file to run one or multiple application instances, *Rim*'s GoLang files to generate requests, Telegraf agent Docker image, Telegraf agents' configuration file.
- *Connections*: `jaeger-agents` connect to `M2:14267` to report spans to `jaeger--collector` and to `M2:2042` to report spans to `kaiju-collector`, telegraf agents connect to `M2:9876` to report metrics to `kaiju-collector` event socket (*JSON*) and to `M3:8086` to send data to the InfluxDB instance

[1] Image available on DockerHub https://hub.docker.com/r/marioscrock/rim/tags/
[2] InfluxDB https://www.influxdata.com/time-series-platform/influxdb/
[3] Telegraf https://www.influxdata.com/time-series-platform/telegraf/
[4] Docker https://www.docker.com
[5] Docker-compose https://docs.docker.com/compose/
[6] Amazon AWS https://aws.amazon.com

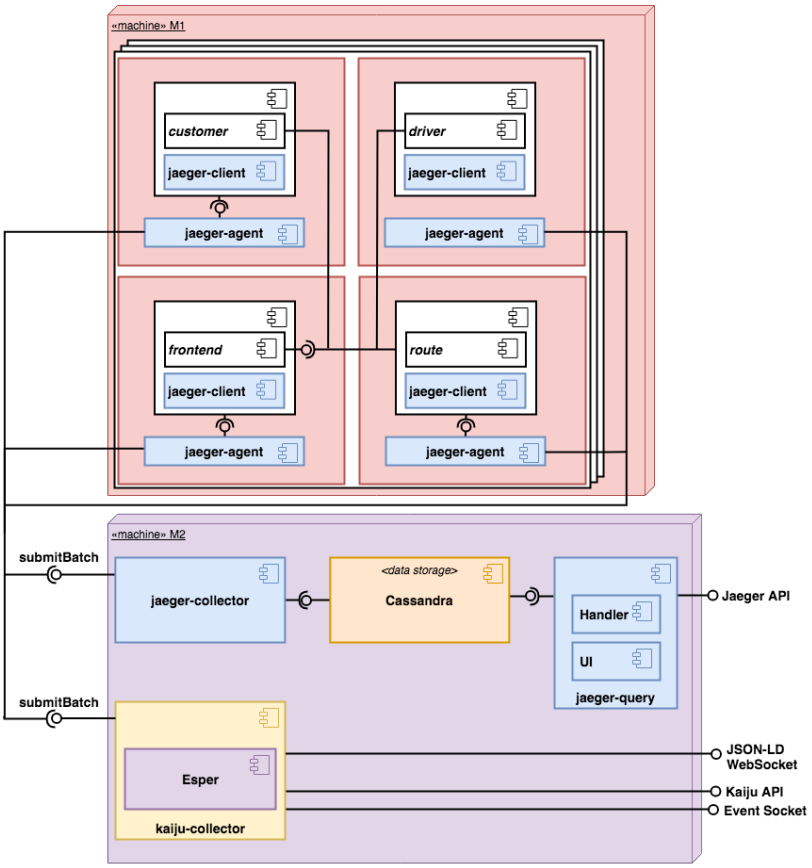

Fig. 6.1 Diagram showing deployment on machines *M1* and *M2*.
M1 hosts *Rim* (one or multiple instances of modified HotR.O.D. application and GoLang scripts to generate requests) and *M2* hosts instances of *Kaiju* and *Jaeger*. Both the machines also host Telegraf agents not reported in the Figure.

- *Expose*: at `8080` the HotR.O.D. UI and the `/dispatch` endpoint (plus other ports if multiple instances)

M2: *Kaiju* and *Jaeger*

- *Instance type*: 1 Linux instance `m4.2xlarge` (8 vCPU | 32 GiB Mem)
  In order to guarantee at least one core to each component.

- *Requirements*: Docker, docker-compose `v2`,
  Docker images (`jaegertracing/jaeger-collector:1.5`, `cassandra:3.9`, `jaegertracing/jaeger-cassandra-schema:1.5`, `jaegertracing/jaeger-query:1.5`, `kaiju-collector`), docker-compose file to

run *Jaeger* and *Kaiju*, Telegraf agent Docker image, Telegraf agents' configuration file, Python `v3`, Python scripts used for evaluation.

- *Connections*: telegraf agents connect to `M3:8086` to send data to the InfluxDB instance
- *Expose*: at `2042` `kaiju-collector` accepts incoming spans via TChannel, at `14267` `jaeger-collector` accepts incoming spans via TChannel, at `9876` `kaiju-collector` accepts incoming *observations* via socket (*JSON*), at `9278` `kaiju-collector` exposes its API, at `4567/streams/jsonTraces` `kaiju-collector` exposes the RDF stream through web socket, at `16686` `jaeger-collector` exposes its API and UI

M3: *InfluxDB*

- *Instance type*: 1 Linux instance `t2.micro` (1 vCPU | 1 GiB Mem)
- *Requirements*: InfluxDB `v1.6`
- *Expose*: at *8086* InfluxDB exposes its API

6.2 Kaiju vs Jaeger

In this section, we report the evaluation made to compare *Kaiju* and *Jaeger*. In particular, we would like to evaluate the advantages and drawbacks of a stream-oriented approach and to show *Kaiju* can provide same expressiveness as *Jaeger* in processing request-scoped *observations*. Indeed, *Kaiju* and *Jaeger* cannot be compared side by side since they approach the problem under two different paradigms: *Jaeger* stores trace data collected to provide a static analysis and the possibility of querying collected trace data whilst, *Kaiju*, offers a dynamic analysis of data that, if not explicitly stored, are discarded after being processed.

However, to pose *Kaiju* on a baseline with respect to state-of-the-art end-to-end tracing systems, we would like to compare the expressiveness of the two approaches in processing the same type of data.

Jaeger offers a visualisation layer for trace data and an API offering methods to ask predefined queries to the storage (in our case *Cassandra*):

- Get all operation names for a given service
- Get all service names

- Get all spans of a traces given its `traceId`
- Get the `traceId` of all traces fulfilling a set of provided parameters (parameters that can be passed are *service name*, *operation name*, *tag key* or a *JSON key-value map* of tags that should be present in the trace, *limit* to set the maximum number of traces reported, *minimum* and *maximum duration* of traces reported, *start time* and *end time* timestamps to define an interval of interest, *lookback* to specify directly an interval)
- Get the pairs representing services interacting within traces stored (an *end timestamp*, a *lookback* and a *service name* parameter can be specified)

To show *Kaiju* can offer the same expressiveness we set the comparison at the API level, showing than *Kaiju* could substitute *Jaeger* transparently from the user perspective. Queries in Esper, also referred to as statements, are continuous, so they are installed and executed until removed. However, it is possible to express fire-and-forget queries against named windows and tables. Tables offer a sort of in-memory relational database, while, in case of named windows, we have structured data, but we must specify an interval of retention for incoming data (configurable parameter when launching the `kaiju-collector`).

In particular, the API methods are implemented in `kaiju-collector` exploiting the concept of *prepared queries* in *Esper*. Indeed, it is possible to express parametric fire-and-forget queries (through the ? operator) that allow the engine to optimise their execution once fired with a set of parameters. An example is provided in Listing 6.1.

```
1 select span.spanId as spanId, span.operationName as operationName,
2   span.getLogs() as logs
3 from SpansWindow
4 where span.getLogs().anyOf(l ⟹ l.getFields().anyOf(f ⟹ f.key = ?))
```

Listing 6.1 Example of prepared query in EPL.

As a steam processor, *Kaiju* is not meant to store data but to process them on the fly. We implement these methods to show how *Kaiju* can offer the same expressiveness provided by the *Jaeger* API.

- `GET /api/traces/all`
  Returns all `traceIds` related to spans currently retained by the *Esper* Engine.
- `GET /api/traces?service=<service>`
  Returns all spans for a given `serviceName` currently retained by the *Esper* Engine.
- `GET /api/traces/:id`
  Returns all spans for a given `traceId` currently retained by the *Esper* Engine.

- `GET /api/dependencies/:traceId`
  Return a set of tuples (`serviceFrom`, `serviceTo`, `numInteractions`) representing interactions between services in a given trace.

The methods reported are the basic methods needed to provide a visualisation layer for traces and a dependency graph. However, we also add another method enabling any arbitrary fire-and-forget queries on named windows and tables to test the possibility of implementing the other methods offered from the *Jaeger* API.

- `POST /api/query?query=<query>`
  Executes the fire and forget query sent.

As pointed out in Section 4.2, the main advantage of a stream processing engine is the possibility to process high volumes of data with low latencies. For this reason, to provide a more fair evaluation of the two pipelines we tested *Jaeger* and *Kaiju* pulling data from the provided APIs.

6.2.1 Configurations

We set up the environment described in Section 5.1 through the `docker-compose` files[7] and we instantiate two different load tests. We started using the `makerequests.js` UI to build a uniform distribution *U1* with the following parameters:

- 5000 requests
- seed = "EXP"
- Time interval = 100 seconds

Then, two different load tests are created as follows:

- **bigLoad** firing U1 uniformly over three HotR.O.D. instances optimized to reduce requests latencies
- **bigLoad+** firing two times U1, with the second one delayed of 50 seconds (both uniformly distributed over three HotR.O.D. instances optimized to reduce requests latencies)

[7] *Rim* https://github.com/marioscrock/Rim/blob/master/hotrod-docker-compose.yml
Kaiju and *Jaeger* https://github.com/marioscrock/Kaiju/blob/master/kaiju-docker-compose.yml

To optimize HotR.O.D. instances we set the following optimizations (compare Table 5.1): (i) –fix-route-worker-pool-size=1000, (ii) –fix-route-calc-delay=1ms, (iii) –fix-db-query-delay=1ms, (iv) –fix-disable-db-conn-mutex, (v) –fix-redis-get-delay=1ms, and (vi) –fix-redis-find-delay=1ms.

To make this test we also remove limits on queue in *Jaeger* to avoid dropping spans through the `-processor.jaeger-compact.server-queue-size` flag in `jaeger-agent` and the `-collector.queue-size` in `jaeger-collector`.

Therefore, we run the two load tests providing different values for the following parameters:

- **Retention time** of *Kaiju* API named windows (can be specified with the EPL syntax for time)
- **Lookback** interval (in seconds) in querying *Jaeger* API
- **Limit** parameter in querying *Jaeger* API

To evaluate the two APIs we exploit two Python scripts `jaegerTiming.py` and `kaijuTiming.py`. These two scripts repeatedly call *Jaeger* and *Kaiju* APIs to obtain all the spans of traces executing the service *frontend*. Moreover, they record timings at which each span is returned for the first time. When the APIs repeatedly do not return any trace, i.e. traces in *Jaeger* storage are too "old" with respect to the provided *lookback* or *Kaiju* named window is empty, they return.

Listing 6.2 reports the commands to run the experiments. In the comments for each command, it is specified the machine on which it must be executed. The three parameters are reported with place-holders, and machines' IP addresses are substituted with machine names.

```
1  # M1 Launch Jaeger and Kaiju
2  sudo RETENTION_TIME="<retention_time>" docker-compose -f kaiju-docker-compose.yml up
3
4  # M2 Launch 3 HotR.O.D. instances
5  # We assume optimizations are explicited in the docker-compose file
6  sudo HOTROD_INSTANCE="hotrod1" JAEGER_COLLECTOR_ADDRESS="M2:14267" KAIJU_ADDRESS="M2:2042"
       HOST_PORT_FRONTEND=8080 docker-compose -f hotrod-docker-compose.yml -p hotrod1 up
7  sudo HOTROD_INSTANCE="hotrod2" JAEGER_COLLECTOR_ADDRESS="M2:14267" KAIJU_ADDRESS="M2:2042"
       HOST_PORT_FRONTEND=8090 docker-compose -f hotrod-docker-compose.yml -p hotrod2 up
8  sudo HOTROD_INSTANCE="hotrod3" JAEGER_COLLECTOR_ADDRESS="M2:14267" KAIJU_ADDRESS="M2:2042"
       HOST_PORT_FRONTEND=8091 docker-compose -f hotrod-docker-compose.yml -p hotrod3 up
9
10 # M2 bigLoad
11 # This command allows to uniformly fire U1 on the 3 instances (endpoints at ports 8080, 8090, 8091)
12 go run makeRequests.go exp1.go
13 # M2 bigLoad+
14 go run makeRequests.go exp1.go & sleep 50; go run makeRequests.go exp1.go
```

```
15
16 # M1
17 JAEGER_LIMIT=<jaeger_limit> JAEGER_LOOKBACK_S=<jaeger_lookback> python jaegerTiming.py \& python
       kaijuTiming.py
```

Listing 6.2 Configuration for the evaluation of *Kaiju* and *Jaeger*

6.2.2 Results

In this section, we report the results obtained discussing how the parameters impact the performances measured at APIs. We are going to comment results through two types of plots:

- A *pie* plot, reporting for *Kaiju* and *Jaeger* the percentages of spans/traces observed and lost.
- A plot showing the number of spans observed over time by the two APIs (blue for *Kaiju*, red for *Jaeger*). Each point represents the total number of spans observed after a given request to the API, and we overlapped a grey representation of requests made. Therefore, this plot allows also visualising delays of the two APIs in observing spans. The plateaus at the end of *Kaiju* and *Jaeger* distributions are related to the final API requests, made by the two scripts, returning an empty response.

bigLoad

We start considering the *bigLoad* test.

As showed in Figure 6.2, selecting a *retention time* of one minute in *Esper* almost all spans are not observed from the API. This problem is due to the fact that each request made to the API returns a considerable amount of data slowing down the API and causing the named windows to shift forward before another request can be made. However, also *Jaeger* doesn't perform better for the same reason.

In Figure 6.3 we show the improvement obtained reducing the *retention time* of *Kaiju* drastically from 1 minute to 1 second. *Kaiju* observes almost any trace, it loses some spans, but it outperforms *Jaeger*. We tried intermediate values between 1 second and 1 minute for the *retention time* of *Kaiju*, but the lower one is the one performing better.

Given this results, we also made tests reducing the *lookback* for *Jaeger*, but this reduces the processing time for each request and increases the frequency of our script requests causing too many read calls to *Cassandra*. Cassandra is deployed on a single node in our deployment, and since it is already serving a high number of insert request, it returns read errors (as shown in Figure 6.4).

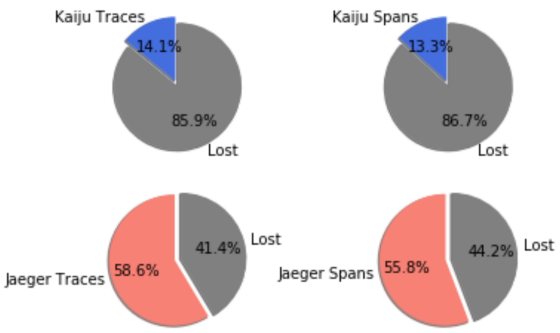

Fig. 6.2 Percentages of spans/traces observed and lost: bigLoad, retention time 1min, lookback 60s, no limit.

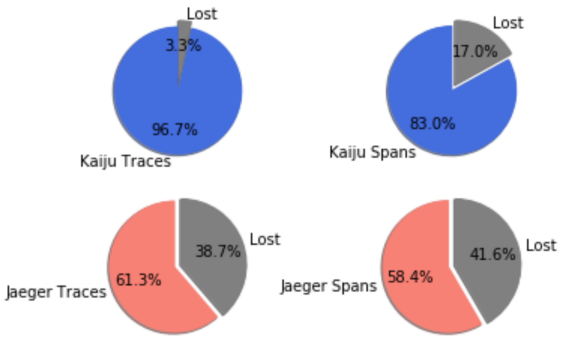

Fig. 6.3 Percentages of spans/traces observed and lost: bigLoad, retention time 1sec, lookback 60s, no limit.

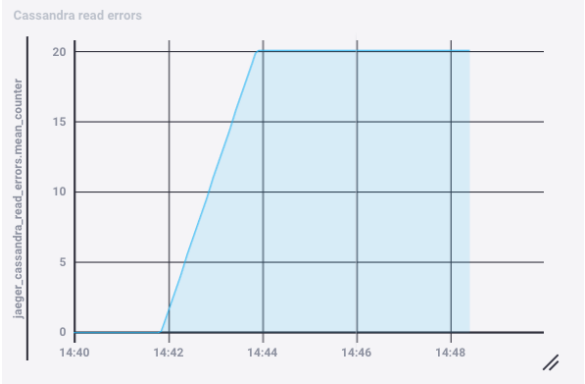

Fig. 6.4 Read errors in Cassandra.

We also tried to modify the *limit* parameter in *Jaeger*, not set previously. We get a slight improvement for *Jaeger* but the final number of spans observed from the *Kaiju* API is still higher, as shown in Figure 6.5. Reducing, even more, the *limit* parameter we encounter the same problem described in reducing the *lookback* parameter.

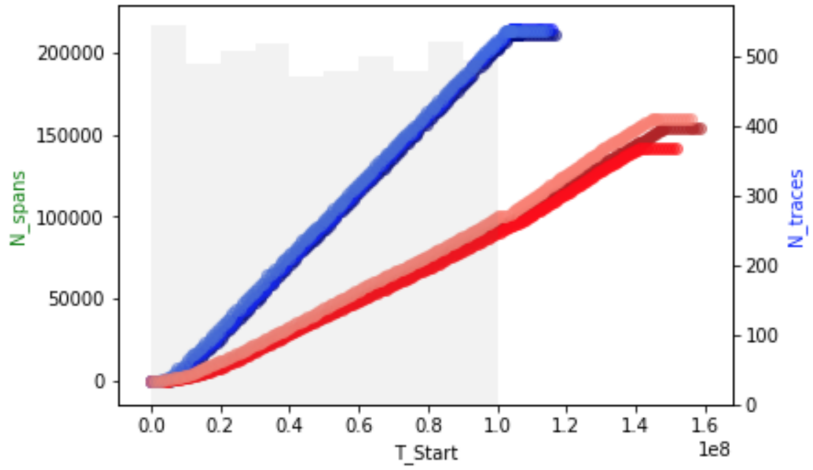

Fig. 6.5 Number of spans observed over time: bigLoad, retention time 1sec, lookback 60s, limit 100.

Performances in *Jaeger* improves if we increase the *lookback* parameter, as shown in Figure 6.6 and in Figure 6.7. This causes less but higher-latency calls to the *Jaeger* API allowing to observe a higher number of spans with respect to *Kaiju* but introducing a *lookback* delay.

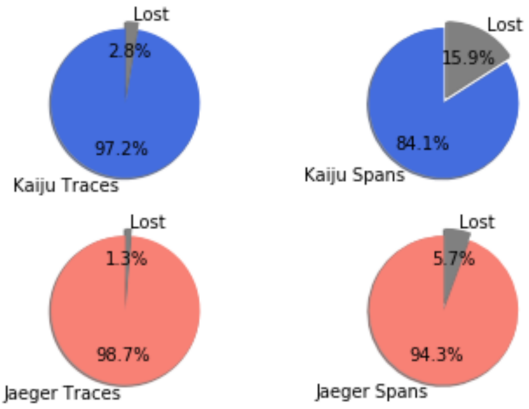

Fig. 6.6 Percentages of spans/traces observed and lost: bigLoad, retention time 1sec, lookback 120s, no limit.

Trying to set the *limit* parameter in *Jaeger* and considering the same *lookback*, we also obtain for *Jaeger* a high number of low latencies requests. However, as shown in Figure 6.8, the *limit* does not allow looking back to recover the same amount of spans as in the previous case, and in the end, *Kaiju* APIs observes a higher number of spans.

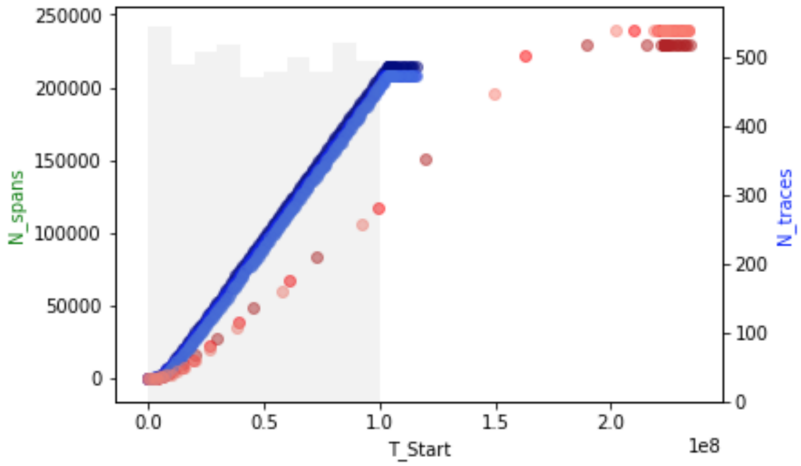

Fig. 6.7 Number of spans observed over time: bigLoad, retention time 1sec, lookback 120s, no limit.

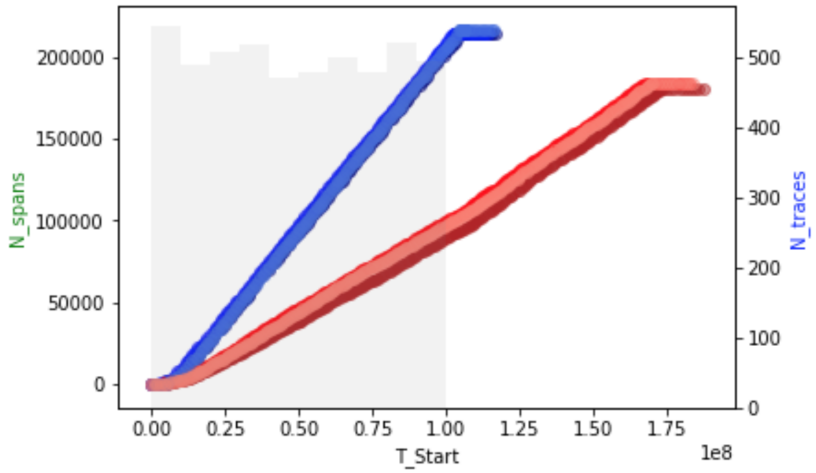

Fig. 6.8 Number of spans observed over time: bigLoad, retention time 1sec, lookback 120s, limit 100.

bigLoad+

The load test *bigLoad+* allows testing the two API with respect to an increasing load.

We consider the best case for *Jaeger* and *Kaiju* observed with the *bigLoad* test and so, we set the *retention time* to 1 sec and the *lookback* parameter to 120s. As shown in Figure 6.9, when the load increases the slope for *Kaiju* changes since fewer spans are reported in the same interval (higher latencies for each call). However, the little *retention time* for *Kaiju* allows to immediately increase frequencies of requests when the load test decreases. *Jaeger*, instead, increases latencies and doesn't manage in the end to observe the same amount of spans of *Kaiju* (Figure 6.10).

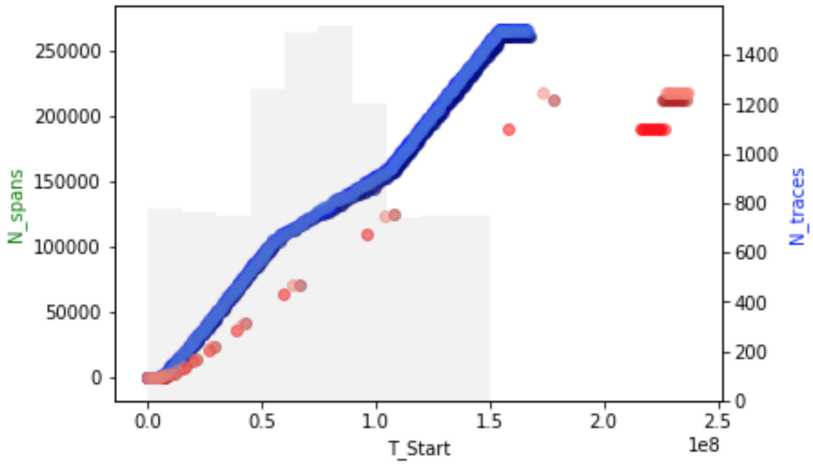

Fig. 6.9 Number of spans observed over time: bigLoad+, retention time 1sec, lookback 120s, no limit.

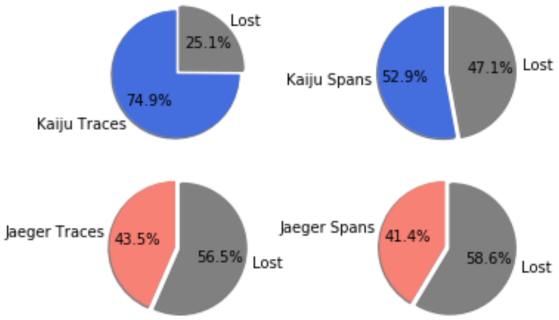

Fig. 6.10 Percentages of spans/traces observed and lost: bigLoad+, retention time 1sec, lookback 120s, no limit.

Setting the *limit* parameter we do not have any significative improvement in *Jaeger* (Figure 6.11). To improve percentages of spans observed from the *Jaeger* API we need to increase the *lookback* causing higher delays in the timing at which the spans are observed (Figure 6.12).

Discussion

With this experiment, we highlight the **indexing overhead** introduced by storing data before processing them and the effectiveness of *Kaiju* in handling high volumes of data. Using *Kaiju*, as we have done to implement the API, is similar to implement a *Jaeger* pipeline with in-memory storage. In both cases, it is possible to avoid index-based storage latencies and to access quickly data gathered, but *Kaiju* offers a lot of other potentialities processing data dynamically.

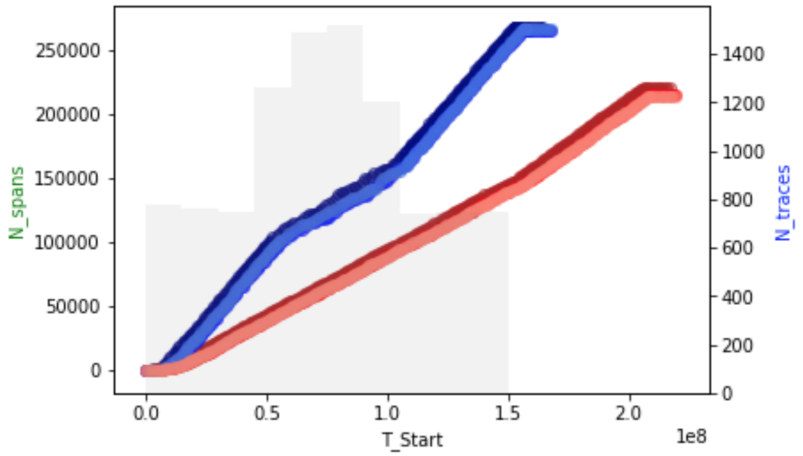

Fig. 6.11 Number of spans observed over time: bigLoad+, retention time 1sec, lookback 120s, limit 100.

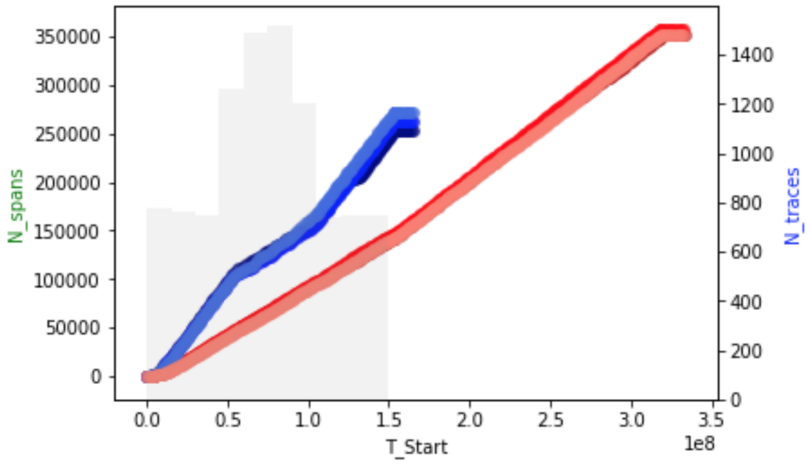

Fig. 6.12 Number of spans observed over time: bigLoad+, retention time 1sec, lookback 180s, limit 100.

To conclude we can claim *Kaiju* can be effectively used to process request-scoped *observations* since it can provide fast access to data through an API with the same expressiveness of state-of-the-art end-to-end tracing systems. Moreover, it allows processing data as soon as they arrive, allowing anyway to store data at a later stage (e.g., it allows to implement a posteriori sampling exposing to the API, or storing, only interesting traces).

In this section, we set a baseline to evaluate *Kaiju*, while in the next ones, we will investigate the advantages offered by the dynamic processing of request-scoped *observations*.

6.3 Processing the stream

In this section, we focus on the evaluations of *Kaiju* potentialities in processing request-scoped *observations*. We decided to make this evaluation against the set of use cases identified in [54] for distributed tracing:

- **Anomaly detection**: use case related to the identification and debugging of problems related to unusual workflows rarely manifesting.
- **Diagnosing steady-state problems**: use case related to the identification and debugging of problems present in the workflow structure. These problems are not anomalies occurring rarely but issues limiting performances of the system.
- **Distributed profiling**: use case related to the identification of slow components or functions also with respect to the inputs provided.
- **Resource usage attribution**: use case related to the attribution of resource usage to the client or request that originally caused it.
- **Workload modeling**: use case related to the creation of workload models representing work-flows trends between components and services.

In evaluating *Kaiju* we exploit *Rim*. Indeed, for each use case, we first show how *Rim* can reproduce a related issue and then we discuss if and how *Kaiju* can address it.

6.3.1 Anomaly detection

Rim

We can configure *Rim* as follows to simulate the presence of anomalies. We deploy: (i) two instances of HotR.O.D. with the default values and the flag `-fix-disable-db-conn-mutex` disabling the mutex, and (ii) a third instance of HotR.O.D. with the default values and the mutex enabled. We execute a uniform workload of requests on the first two instances, and then with predefined timings, we send a little number of requests to the third instance. In this way, we can simulate a misconfigured connection pool guaranteeing for some types of requests only one connection at the time and, thus, simulating an abnormal increase of latency for a small percentage of requests.

Moreover, for this use case, we report in Listing 6.3 the configuration in the deployment described in Section 6.1 and the obtained results. We started using the `makerequests.js` UI to build 2 uniform distributions with the following parameters:

- *U1*: 5000 requests, seed = "EXP", Time interval = 100 seconds
- *U2*: 50 requests, seed = "EXP", Time interval = 10 seconds

We then implement the GoLang scripts to fire *U1* on the first two instances and *U2* on the misconfigured one creating a workload test as reported in Figure 6.13 (in green requests made to the first two instances, in red requests made to the third one).

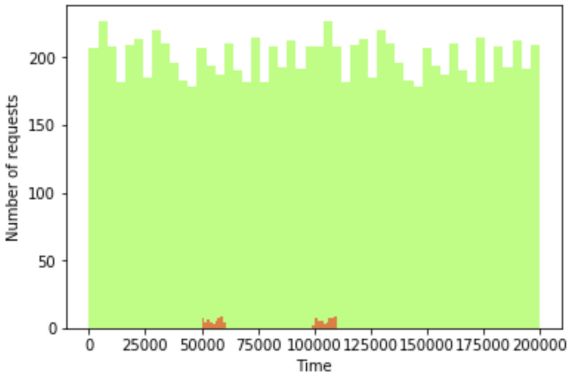

Fig. 6.13 Workload applied in anomaly detection experiments.

```
1  # M1
2  sudo RETENTION_TIME="1min" docker-compose -f kaiju-docker-compose.yml up
3
4  # M2
5  sudo HOTROD_INSTANCE="hotrod1" JAEGER_COLLECTOR_ADDRESS="M2:14267" KAIJU_ADDRESS="M2:2042"
       HOST_PORT_FRONTEND=8080 docker-compose -f hotrod-docker-compose.yml -p hotrod1 up
6  sudo HOTROD_INSTANCE="hotrod2" JAEGER_COLLECTOR_ADDRESS="M2:14267" KAIJU_ADDRESS="M2:2042"
       HOST_PORT_FRONTEND=8090 docker-compose -f hotrod-docker-compose.yml -p hotrod2 up
7  sudo HOTROD_INSTANCE="hotrod3" JAEGER_COLLECTOR_ADDRESS="M2:14267" KAIJU_ADDRESS="M2:2042"
       HOST_PORT_FRONTEND=8091 FIX_DISABLE_DB_CONN_MUTEX="--fix-disable-db-conn-mutex" docker-compose -f
       hotrod-docker-compose.yml -p hotrod3 up
8
9  #M2
10 go run makeRequests.go aU1.go & sleep 50; go run makeRequests.go aU2.go & sleep 50; go run
       makeRequests.go aU2.go & go run makeRequests.go aU1.go
```

Listing 6.3 Configuration for the anomaly detection experiment.

Kaiju

Anomaly detection requires rapid processing of *observations* to guarantee low latency of detection and the possibility to express conditions triggering anomalies. *Kaiju* can offer both providing stream processing and offering an expressive query language. Moreover, it can handle a large amount of data requiring less strict sampling strategies and for this reason more suitable to detect anomalies occurring sporadically.

We will now discuss how we can address the issue proposed in *Rim* with *Kaiju*. We need to monitor anomalies in operation latencies to detect the misbehaving service and operation and to report the spans showing the issue. To this purpose we can create a table to keep track of the mean and variance of each operation, updated following the Welford's Online algorithm through the *on-merge-update* syntax of *Esper*. We express this logic in EPL, but it is also possible to implement it through a custom Java function aggregating values with the same logic and then use it in the statement.

Listing 6.4 creates the *MeanDurationPerOperation* table selecting service name and operation name as primary keys.

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

Listing 6.4 Create table *MeanDurationPerOperation* in EPL.

As shown in Listing 6.5, on each incoming span we check if the service name and operation name are already present in the table, if true we update the corresponding row following the algorithm, otherwise, we initialize the row. We also need to select a *resetValueInt*, showed in the listing as a place-holder, since coefficients of the algorithm are monotonically increasing and then we need to reset them at some point.

```
1 on SpansWindow s
2 merge MeanDurationPerOperation m
3 where s.serviceName = m.serviceName and s.span.operationName = m.operationName
4 when matched and counter ≤ <resetValueInt>
5     then update set counter = (initial.counter + 1), meanDuration = (initial.meanDuration +
          ((span.duration − initial.meanDuration)/counter)), m2 = (initial.m2 + (span.duration −
          meanDuration)∗(span.duration − initial.meanDuration))
6 when matched and counter > <resetValueInt>
7     then update set counter = 1, meanDuration = s.span.duration, m2 = 0
8 when not matched
9     then insert select s.serviceName as serviceName, s.span.operationName as operationName,
          s.span.duration as meanDuration, 0 as m2, 1 as counter
```

Listing 6.5 On-merge-update of table *MeanDurationPerOperation* in EPL.

Given the table described above, we need to express a rule to identify anomalous spans. Listing 6.6 implements the so-called *three-sigma rule*, that assumes a Gaussian distribution of samples and detects tails of the distribution fulfilling the equation $(duration - meanDuration) > 3 * stdDev$. It is important to notice that the detection of anomalies can also rely on a custom data function, for example, reporting a dynamic threshold computed on historical data.

```
1 insert into HighLatency3SigmaRule
2 select traceIdToHex(span.traceIdHigh, span.traceIdLow) as traceId, Long.toHexString(span.spanId) as
      spanId, serviceName, span.operationName as operationName, span.startTime as startTime,
      span.duration as duration, p.hostname as hostname
3 from SpansWindow as s join ProcessesTable as p
```

```
4 where s.hashProcess = p.hashProcess and (span.duration - MeanDurationPerOperation[serviceName,
      span.operationName].meanDuration) > 3 *
      java.lang.Math.sqrt((MeanDurationPerOperation[serviceName, span.operationName].m2) /
      (MeanDurationPerOperation[serviceName, span.operationName].counter))
```

Listing 6.6 Rules generating *HighLatency3SigmaRule* events in EPL.

Once generated the *HighLatency3SigmaRule* stream, we can listen for this type of events marking related traces as to-be-sampled and, thus, reporting traces containing spans with anomalous latency values (as shown in Section 5.2.3). We run the experiment proposed to evaluate the latency of detection and to check only anomalous traces are reported. In Figure 6.14 we show almost all anomalous requests directed to the misconfigured instance are detected in four different executions of the same workload. The number of anomalies detected is not total since first requests get the database connection immediately without the need to wait for the mutex and, therefore, are executed with the same latencies of other operations.

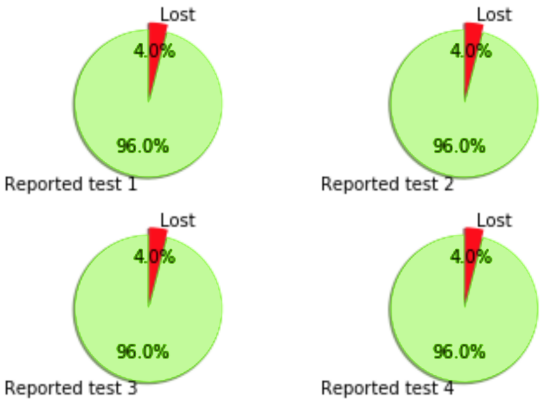

Fig. 6.14 Catched anomalies on total anomalous requests generated.

In Figure 6.15 and in Figure 6.16, instead, we show the distance in time between anomalies introduction and anomalies detection. We here represent all anomalies detected for a specific service name or operation name to check anomalies are reported only in correspondence of requests made to the misconfigured instance. In Figure 6.15 we represent all anomalies detected querying for the operation name *SQL SELECT*, the operation affected by the introduced mutex, while in Figure 6.16 we represent all anomalies detected querying for the service name *customer*, the service performing the operation. We can observe in both cases anomalies are reported (with low latencies) only on anomaly introduction. The exception, related to the anomalies reported at the beginning, reflects the fact that the algorithm is initialised with default values. Therefore, mean and variance need some time to become stable and meaningful.

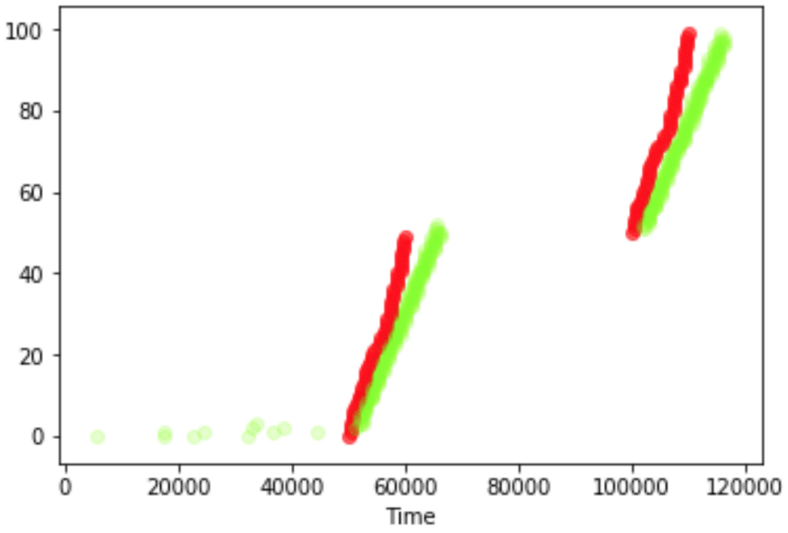

Fig. 6.15 Anomalies reported for operation name *SQL SELECT* on all instances.

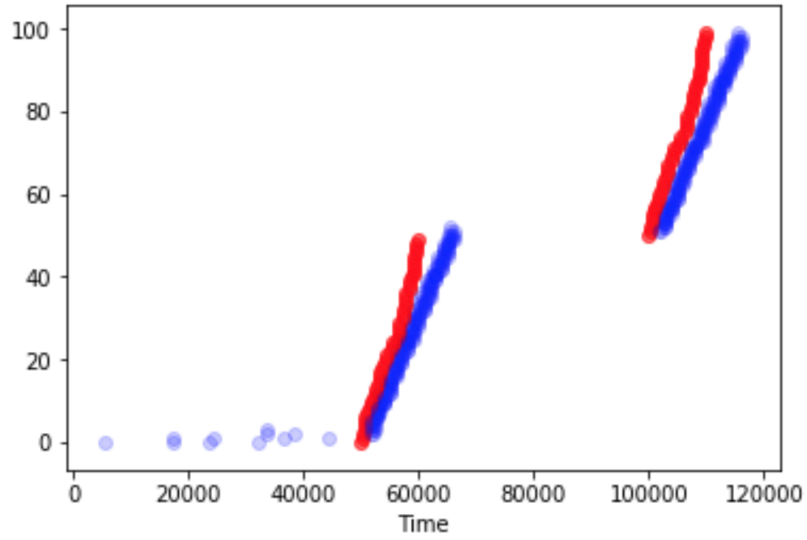

Fig. 6.16 Anomalies reported for service name *customer* on all instances.

To conclude, we can claim *Kaiju* enables rapid detection of anomalies allowing expressiveness in defining rules. Moreover, we can exploit *Kaiju* to further analyse the system behaviour once an anomaly is reported, e.g., in the scenario considered: (i) we can install a statement counting for traces with latency higher than a specific threshold (e.g. three-sigma rule) and grouping them by the process, (ii) we can point out almost all traces are related to the same process, (iii) we query for traces served by the given process, and we inspect them, (iv) from logs of the span we can notice a high number of locked

requests trying to access the DB, and (v) we understand the problem is in the connection pool and we can alert the responsible team.

6.3.2 Diagnosing steady-state problems

Rim

Steady-state problems are related to issues present in almost any trace. One example in *Rim* can be provided launching an instance of HotR.O.D. with the default values for `route-calc-delay` and `route-worker-pool-size`. Indeed, in this case, the worker pool size is too small, and requests made to the route service are not efficiently parallelised.

Kaiju

These type of problems are often easily-discoverable though a visualisation layer, and indeed they are the primary use case addressed from tools like *Jaeger* and *Zipkin*. However, *Kaiju* can be used to compute useful analytics to understand which traces or spans should be investigated to optimise the overall performance, e.g. to discover possible bottlenecks.

For example, Listing 6.7 uses the table in Listing 6.4, aggregating average latencies per operation, to report the TOP-K operations ordered by latency.

In Listing 6.7 the parameter K is a place-holder to be tuned.

```
1 select serviceName, operationName, meanDuration, (m2/counter) as variance, counter
2 from MeanDurationPerOperation
3 output snapshot every 5 seconds
4 order by meanDuration desc
5 limit <K>
```

Listing 6.7 Top-K query on *MeanDurationPerOperation* table in EPL.

To conclude, *Kaiju* can be useful to point out bottlenecks and to compute analytics on system performances, but for some use-cases, it is more relevant to investigate traces through a visualisation layer. For example, not exploited parallelism cannot be easily identified through a statement, whereas a static analysis of a Gantt trace representation can easily show operations executed in a sequence that can be parallelised.

6.3.3 Distributed profiling

Rim

Issues addressed by distributed profiling are often related to specific stacks and inputs causing functions or components to be slow. In *Rim* we can, for example, simulate an

issue of this type as follows: (i) one instance of HotR.O.D. with an optimization to reduce `db-query-delay` (e.g., `-fix-db-query-delay=1ms`), and (ii) one instance of HotR.O.D. with default values. We can then redirect requests between the two instances based on the *customer id* of the request to simulate data about some customers are cached (simulated by the optimised instance), and others are not cached, and the database needs to be accessed (simulated by the instance with the default value).

Kaiju

Distributed profiling is about computing analytics on request-scoped *observations* and for this reason it can be addressed effectively by *Kaiju*.

Considering the issue proposed in *Rim*, we can profile trace latencies grouping them per customer to determine if they are similar or different. As shown in Listing 6.8, we can aggregate latencies of traces asking for the root span (parentSpanId=0) and then exploit the contained events syntax (`[]` parentheses) to query logs and group them by *customerId*.

```
1 select customerId, avg(duration) as meanDuration, stddev(duration) as stdDevDuration
2 from Span(parentSpanId = 0)[select duration, l.getFields().firstOf(f ⟹ f.key =
      'customer_id').getVStr() as customerId from logs as l where l.fields.anyOf(f ⟹
      f.key='customer_id')]
3 group by customerId
4 output snapshot every 5 seconds
5 order by meanDuration desc
```

Listing 6.8 Average latency of traces per customer in EPL.

With this statement, we can detect if requests related to a specific customer performs differently. Moreover, considering the issue proposed, we can *count* the number of requests for each customer and determine if it would be beneficial to force caching for specific customers data. Similarly we can aggregate data for each type of parameter characterising the request enabling profiling.

6.3.4 Resource usage attribution

The resource usage attribution use case exploits a key aspect about querying request-scoped *observations*. It is based on the possibility to enable request-wise correlation between events happening in the system. In this way, it is possible to attribute resources usage in a component taking into account the entire request generating that usage.

Rim

In *Rim*, considering a default instance of HotR.O.D., the `route` service tags in its span the time spent by the CPU to perform the computation. This data can be used to compute the total usage in a specific interval but also allows aggregating this value taking into account the request generating that usage. For example, we can group usage by `customerId` or `sessionId` of the request. These are parameters tagged or logged in others spans of the request but retrievable exploiting request-wise querying.

Kaiju

In *Kaiju* we can join *Span* streams and exploits the contained-event selection to address the issued proposed. Listing 6.9 joins for each trace the span related to the operation `HTTP GET /customer`, selecting from its logs the `customerId`, and the one related to the `HTTP GET /route` operation, selecting the CPU usage. In this way, we can group data retrieved per `customerId` and attribute CPU usage on a per customer basis.

```
1 select customerId, sum(timeCPU) as timeCPURouteCalcperCustomerId
2 from Span(operationName = "HTTP GET /customer")[select traceIdHigh, traceIdLow,
      l.getFields().firstOf(f ⟹ f.key ='customer_id').getVStr() as customerId from logs as l where
      l.fields.anyOf(f ⟹ f.key='customer_id')]#time(<retentionTime>) as s1,
3 Span(operationName = "HTTP GET /route")[select traceIdHigh, traceIdLow, l.getFields().firstOf(f ⟹
      f.key ='time').getVDouble() as timeCPU from logs as l where l.fields.anyOf(f ⟹ f.getVStr() =
      'RouteCalc')]#time(<retentionTime>) as s2
4 where s1.traceIdHigh = s2.traceIdHigh and s1.traceIdLow = s2.traceIdLow
5 group by customerId
6 output last every 10 seconds
```

Listing 6.9 CPU usage in `route` service per `customerId` in EPL.

Similarly, Listing 6.10 joins for each trace the span related to the operation `Driver::findNearest`, selecting from its logs the `sessionId`, and the one related to the `HTTP GET /route` operation, selecting the CPU usage. In this way, we can group data retrieved per `sessionId` and attribute CPU usage on a per session basis.

```
1 select sessionId, sum(timeCPU) as timeCPURouteCalcperSessionId
2 from Span(operationName = "Driver::findNearest")[select traceIdHigh, traceIdLow,
      l.getFields().firstOf(f ⟹ f.key='value').getVStr() as sessionId from logs as l where
      l.fields.anyOf(f ⟹ f.getVStr()='session')]#time(<retentionTime>) as s1,
3 Span(operationName = "HTTP GET /route")[select traceIdHigh, traceIdLow, l.getFields().firstOf(f ⟹
      f.key ='time').getVDouble() as timeCPU from logs as l where l.fields.anyOf(f ⟹
      f.getVStr()='RouteCalc')]#time(<retentionTime>) as s2
4 where s1.traceIdHigh = s2.traceIdHigh and s1.traceIdLow = s2.traceIdLow
5 group by sessionId
6 output last every 10 seconds
```

Listing 6.10 CPU usage in `route` service per `sessionId` in EPL.

This powerful possibility is complementary to the concept of baggage proposed in OpenTracing. The baggage exploits the propagation of metadata operated by tracing instrumentation to share data between components across request flow. This method is beneficial for some functionalities but can be a bad choice for others as resources usage attribution. Indeed, in the example proposed, it would have been possible to put `customerId`, and `sessionId` in the baggage and allow the `route` service to directly log CPU usage on a per customer and per session basis. However, this has one main drawback. To extend the set of parameters on which we can aggregate, we will need to propagate them in the baggage modifying code. It is better to log the CPU usage without any pre-aggregation and then correlate it with parameters provided by other spans of the same trace.

Pivot Tracing [47] addresses this problem through baggage and enabling dynamic instrumentation of components, *Kaiju* exploits static instrumentation, avoid baggage overhead and allows to correlate spans when being processed.

6.3.5 Workload modelling

Rim

To model an unusual workload in *Rim* we can consider three instances of HotR.O.D., a uniform workload targeting two of them and a gaussian workload targeting the other instance. In this way, assuming IP-based load balancing between instances we simulate a peak of requests from a particular region to be rapidly detected and re-directed among other instances.

Kaiju

In *Kaiju* we can process data in several ways to model workloads.

One example, as described in Section 6.2, is to construct a service dependency graph as done by *Jaeger* reporting dependencies among services observed within traces. Similar analytics can be useful, for example, in large and highly distributed applications to optimise network communications between components. To build the graph we need to retrieve span references within traces. Listing 6.11 creates a named window to store dependencies between services within a trace (`<retentionTime>` is a place-holder and

should be substituted with an appropriate interval).

```
1 create window DependenciesWindow#time(<retentionTime>) (traceIdHexFrom string, spanIdFrom long,
      traceIdHexTo string, spanIdTo long)
```

Listing 6.11 Create named window to store dependencies between services within a trace in EPL.

Listing 6.12 shows how to insert dependencies in the built named window from the *Span* stream.

```
1 insert into DependenciesWindow
2 select traceIdToHex(s.traceIdHigh, s.traceIdLow) as traceIdHexTo, s.spanId as spanIdTo,
      traceIdToHex(s.r.traceIdHigh, s.r.traceIdLow) as traceIdHexFrom, s.r.spanId as spanIdFrom
3 from Span[select spanId, traceIdLow, traceIdHigh,* from span.references as r] s
```

Listing 6.12 Detection of *Spans*' references within a trace in EPL.

Another example in *Kaiju* is the possibility to profile workload over the day/week setting sliding windows (e.g., to automate resources scaling following workload patterns).

A last example is related to the issue proposed in *Rim*. Through *Kaiju* we can set a gauge counting requests on a per instance basis (Listing 6.13), we can evaluate if the workload is correctly balanced among instances and we can determine instances receiving anomalous number of requests to fix the load-balancing mechanism.

```
1 select hostname, count(*) as counter from Batch[select process.tags.firstOf(t ⟹ t.key =
      'hostname').getVStr() as hostname, * from spans as s where s.parentSpanId =
      0]#time(<retentionTime>)
2 group by hostname
3 order by counter desc
```

Listing 6.13 Gauge counting requests per instance in EPL.

6.3.6 Results

To conclude this section we summarise our evaluation through the Table 6.1.

Use cases	Jaeger	Kaiju
Anomaly detection		
Diagnosing steady-state problems		
Distributed profiling		
Resource usage attribution		
Workload modeling		

Table 6.1 Evaluation of *Kaiju* and *Jaeger* against tracing use cases.

Jaeger pipeline enables a static analysis of traces that can be visualised through the interface and inspected. It mainly addresses the use cases related to the *diagnosis of steady-*

state problems and partially the *workload modelling* offering a services dependencies graph. As shown in Section 6.2, *Kaiju* offers the same expressiveness of *Jaeger*, even if, currently, it cannot offer full coverage for the use cases mentioned for *Jaeger*. Indeed, both use cases requires a visualisation layer, currently not implemented in *Kaiju*, even if, we showed we could provide a suitable API to implement it.

Kaiju, moreover, enables a **dynamic analysis** of traces through continuous querying, fire-and-forget queries on specific windows, event-based pattern rules and detection of complex events. Thanks to the expressiveness of its language and the stream-oriented approach, processing request-scoped *observations* it can then cover all other use cases of tracing: (i) *anomaly detection*, (ii) *distributed profiling*, (iii) *resource usage attribution*.

6.4 Pattern detection

In this section we discuss a further use case enabled extending *Kaiju* to ingest also other types of *observations*, as detailed in Section 5.3. Exploiting the simultaneous processing of multiple streams, and the possibility in EPL to express *detecting rules*, we can enable the following use case:

> **Pattern detection** use case related to the identification of patterns and complex events on system behaviour crossing information coming from different sources and different types of *observations*.

As we have done for previous use cases, we explain how *Rim* can reproduce a set of issues related to the use case, and we show how *Kaiju* can challenge this issues.

Rim

For this use case we propose two different issues in *Rim*:

a) We consider one instance of HotR.O.D. optimizing the `route-worker-pool-size` (e.g. `-fix-route-worker-pool-size=1000`) and another instance with default values. We consider a uniform workload initially directed to the optimized instance, and then redirected to the one not optimized. In this way, we can simulate a deployed commit producing a misconfigured new build of the service.

b) We consider one instance of HotR.O.D. optimizing parameters with the exception of the `route-worker-pool-size` and `route-calc-delay` and an exponential workload targeting it and causing an increment in CPU usage.

Kaiju

Considering Issue *a)* we can assume the *Kaiju*'s event socket receives a new event each time a new commit is deployed. We can then write a statement to generate a *CommitEvent* when this is detected in the *Event* stream generated from the socket (Listing 6.14). We here assume the *CommitEvent* is characterized by the `commit_id` key in its context.

```
1 insert into CommitEvent
2 select timestamp, context('commit_id') as commit, payload('commit_msg') as commitMsg
3 from Event
4 where context.containsKey('commit_id')
```

Listing 6.14 Detect *CommitEvent* in EPL.

We can know declare a pattern that given a *CommitEvent* generates an alert for every *Anomaly* event observed in a specified interval (`within` place-holder in Listing 6.15). In our implementation we made the *HighLatency3SigmaRule* a subtype of the *Anomaly* event testing the pattern is caught in the situation described in Issue *a)*.

```
1 select b.commit, a.*
2 from pattern [b=CommitEvent -> every a=Anomaly where timer:within(<within>)]
```

Listing 6.15 Detect *CommitEvent* and *Anomaly* pattern in EPL.

Considering Issue *b)* we assume, instead, *Kaiju*'s event socket receives metrics from the Telegraf agent collecting *Rim*'s metrics. We can then write a statement to generate a *ProcessCPUHigherThan80* when the CPU usage of one of the HotR.O.D. services exceeded the 80% (Listing 6.16).

```
1 insert into ProcessCPUHigherThan80
2 select hashProcess, process.serviceName as serviceName, hostname,
      Float.parseFloat(fields('usage_percent')) as usagePercent
3 from Metric(name='docker_container_cpu') as m join ProcessesTable as p
4 where m.tags('host') =  p.hostname and Float.parseFloat(fields('usage_percent')) > 80.0
5 output last every 10sec
```

Listing 6.16 Detect *ProcessCPUHigherThan80* in EPL.

We can know declare a pattern that given a *ProcessCPUHigherThan80* generates an alert for every *HighLatency3SigmaRule* event observed in a specified interval (`within` place-holder in Listing 6.17) and having same *hostname*.

```
1 select a.hostname as hostname
2 from pattern [a=ProcessCPUHigherThan80 and b=HighLatency3SigmaRule(hostname = a.hostname) where
      timer:within(<within>)]
```

Listing 6.17 Detect *ProcessCPUHigherThan80* and *HighLatency3SigmaRule* pattern in EPL.

6.5 Processing the RDF stream

In this section, we made an explorative analysis of the benefits of applying RDF Stream Processing to request-scoped *observations*.

Given the deployment described in Section ch6:sec1, we attach a CSPARQL2 Engine to the *Kaiju* JSON-LD web socket, as described in Section 5.4, to query the RDF stream of data gathered.

In Section 6.5.1 and in Section 6.5.2, we consider two use cases requiring complex queries in EPL, and we show examples on how it is possible to solve them with queries in the RSP-QL syntax.

In Section 6.5.3, we discuss a further possibility enabled from RDF Stream Processing. We can bound RDF graphs to higher level concepts, e.g., belonging to the application domain, and query information from request-scoped *observations* without the need of knowing their data format.

6.5.1 Resource usage attribution

To apply RDF Stream Processing to the *resource usage attribution* use case, we consider the same issue reproduced by *Rim* and described in Section 6.3.4. We express in RSP-QL the same queries shown in EPL to provide a comparison in query writing for resource usage attribution.

In Listing 6.18, we show the aggregate CPU usage in `route` service grouped by `customerId`, whilst, in Listing 6.19, we show the aggregate CPU usage in `route` service grouped by `sessionId`.

```
PREFIX tr: <http://polimi.deib/tracing#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

REGISTER RSTREAM <s1> AS
SELECT (?cid AS ?CustomerId) (SUM(?tCalc) AS ?timeCPURouteCalcperCustomerId)
FROM NAMED WINDOW <win> ON <jsonTraces> [RANGE PT10S STEP PT10S]
WHERE {
    WINDOW ?w {
        ?t a tr:Trace .
        ?s a tr:Span ;
            tr:spanOfTrace ?t ;
            tr:operationName "HTTP GET /customer"^^xsd:string ;
            tr:hasLog ?log .
        ?log tr:hasField ?f .
        ?f tr:tagKey "customer_id"^^xsd:string ;
          tr:stringVal ?cid.
        ?s2 a tr:Span ;
            tr:spanOfTrace ?t ;
            tr:operationName "HTTP GET /route"^^xsd:string ;
```

```
20              tr:hasLog ?log2 .
21          ?log2 tr:hasField ?fR ;
22            tr:hasField ?fT .
23          ?fR tr:stringVal "RouteCalc"^^xsd:string .
24          ?fT tr:tagKey "time"^^xsd:string ;
25            tr:doubleVal ?tCalc .
26      }
27  }
28  GROUP BY ?cid
```

Listing 6.18 CPU usage in `route` service per `customerId` in RSP-QL.

```
1  PREFIX tr: <http://polimi.deib/tracing#>
2  PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
3
4  REGISTER RSTREAM <s1> AS
5  SELECT (?sid AS ?SessionId) (SUM(?tCalc) AS ?timeCPURouteCalcperSessionId)
6  FROM NAMED WINDOW <win> ON <jsonTraces> [RANGE PT10S STEP PT10S]
7  WHERE {
8      WINDOW ?w {
9          ?t a tr:Trace .
10         ?s a tr:Span ;
11             tr:spanOfTrace ?t ;
12             tr:operationName "Driver::findNearest"^^xsd:string ;
13             tr:hasLog ?log .
14         ?log tr:hasField ?fs ;
15           tr:hasField ?fsid .
16         ?fs tr:stringVal "session"^^xsd:string .
17         ?fsid tr:tagKey "value"^^xsd:string ;
18           tr:stringVal ?sid .
19         ?s2 a tr:Span ;
20             tr:spanOfTrace ?t ;
21             tr:operationName "HTTP GET /route"^^xsd:string ;
22             tr:hasLog ?log2 .
23         ?log2 tr:hasField ?fR ;
24           tr:hasField ?fT .
25         ?fR tr:stringVal "RouteCalc"^^xsd:string .
26         ?fT tr:tagKey "time"^^xsd:string ;
27           tr:doubleVal ?tCalc .
28     }
29 }
30 GROUP BY ?sid
```

Listing 6.19 CPU usage in `route` service per `sessionId` in RSP-QL.

6.5.2 Workload modeling

To apply RDF Stream Processing to the *workload modelling* use case, we consider the query required to build the service dependency graph of a trace. In Listing 6.20, for each trace, we report all pairs of services interacting between them and the number of interactions present in the trace.

```
PREFIX tr: <http://polimi.deib/tracing#>

REGISTER RSTREAM <s1> AS
SELECT (?tid AS ?traceId) (?sn2 AS ?serviceFrom) (?sn AS ?serviceTo) (COUNT(*) AS ?n_interactions)
FROM NAMED WINDOW <win> ON <jsonTraces> [RANGE PT10S STEP PT10S]
WHERE  {
    WINDOW <win> {
        ?s tr:reference ?s2 .
        ?s tr:spanId ?sid ;
            tr:spanOfProcess ?p .
        ?p tr:serviceName ?sn .
        ?s2 tr:spanOfTrace ?t ;
            tr:spanId ?sid2 ;
            tr:spanOfProcess ?p2 .
        ?t tr:traceId ?tid.
        ?p2 tr:serviceName ?sn2 .
    }
}
GROUP BY ?tid ?sn ?sn2
ORDER BY ?tid
```

Listing 6.20 Number of interactions between each pair of services within a trace in RSP-QL.

6.5.3 Domain-driven debugging

In this section, we would like to highlight the possibilities provided by an ontological representation of request-scoped *observations*. We will show how to enable querying on request-scoped *observations* produced by *Rim*, considering only the domain-specific terminology of HotR.O.D. application, e.g., to compute analytics on drivers, customers and users.

To this purpose, we designed a simple HotR.O.D. ontology representing its application domain (shown in Figure 6.17):

(i) the ***User*** class representing a UI active session,

(ii) the ***Customer*** class representing a customer (and the 4 individuals in *Customer* HotR.O.D. UI),

(iii) the ***Driver*** class representing a driver,

(iv) the ***requestCarTo*** object property relating each user to the customers he requests a car to,

(v) the ***assignedDriver*** object property relating each user to the driver assigned to it,

(vi) the ***driverToCustomer*** object property relating each driver to the customer he should reach,

(vii) the ***userProperty*** data property, super-property of the *sessionID* and *requestID* property,

(viii) the ***customerProperty*** data property, super-property of the *customerID* and *customerName* property,

(ix) the ***driverProperty*** data property, super-property of the *licensePlateNumber* property,

(x) the ***location*** data property relating an entity to a string representing its location.

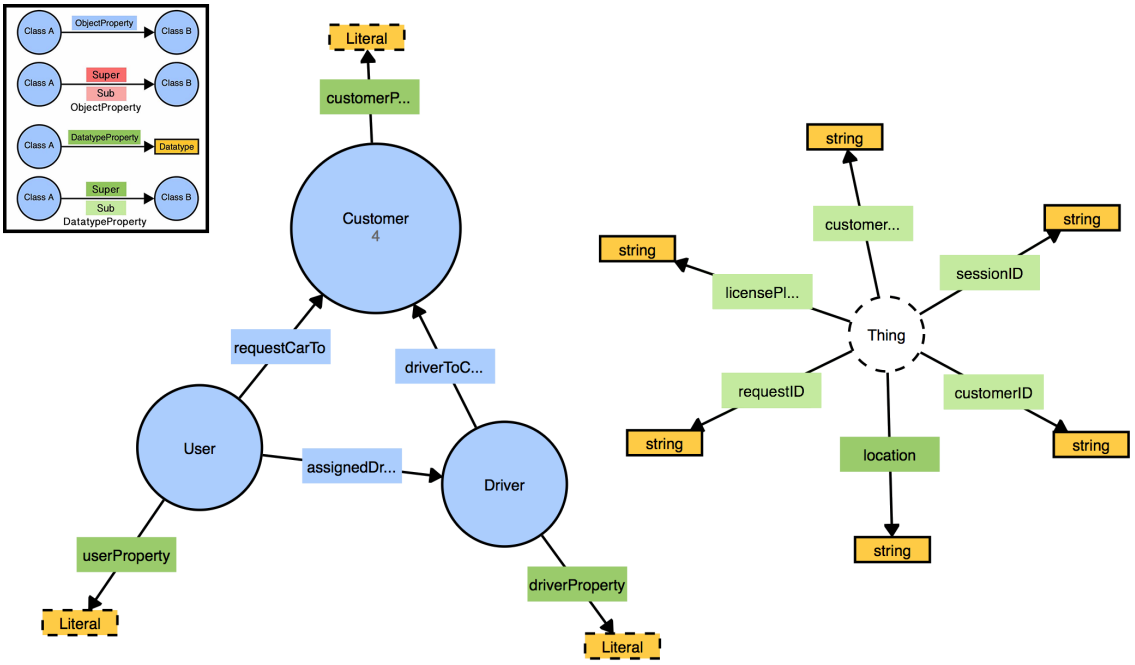

Fig. 6.17 HotR.O.D. ontology.

Therefore, we can exploit the *CONSTRUCT* query type in RSP-QL to generate a stream based on patterns matched in the *Kaiju* stream. In Listing 6.21, we first select data on the user instantiating the request (`sessionId`), data on the customer location requested (`customerId`) and data on the assigned driver (`licensePlateNumber`). Then, we construct an RDF stream modelling the data selected through the HotR.O.D. ontology.

```
PREFIX tr: <http://polimi.deib/tracing#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX ht: <http://polimi.deib/hotrod#>

REGISTER RSTREAM <s1> AS
CONSTRUCT {
    ?user a ht:User;
        ht:sessionID ?sid ;
        ht:assignedDriver ?driver ;
        ht:requestCarTo ?customer .
    ?driver a ht:Driver;
        ht:licensePlateNumber ?did .
}
FROM NAMED WINDOW <win> ON <jsonTraces> [RANGE PT5S STEP PT5S]
WHERE {
    WINDOW <win> {
        ?t a tr:Trace .
        ?s a tr:Span ;
            tr:spanOfTrace ?t ;
            tr:operationName "HTTP GET /dispatch"^^xsd:string ;
            tr:hasLog ?logd ;
            tr:hasLog ?logc .
        ?logd tr:hasField ?fd ;
          tr:hasField ?fdid .
        ?fd tr:stringVal "Dispatch successful"^^xsd:string .
        ?fdid tr:tagKey "driver"^^xsd:string ;
          tr:stringVal ?did .
        ?logc tr:hasField ?fc ;
          tr:hasField ?fcid .
        ?fc tr:stringVal "Getting customer"^^xsd:string .
        ?fcid tr:tagKey "customer_id"^^xsd:string ;
          tr:stringVal ?cid .

        ?s2 a tr:Span ;
            tr:spanOfTrace ?t ;
            tr:operationName "Driver::findNearest"^^xsd:string ;
            tr:hasLog ?log .
        ?log2 tr:hasField ?fs ;
          tr:hasField ?fsid .
        ?fs tr:stringVal "session"^^xsd:string .
        ?fsid tr:tagKey "value"^^xsd:string ;
          tr:stringVal ?sid .
    }

    BIND(URI(CONCAT("http://polimi.deib/hotrod#User", MD5(STR(?sid)))) as ?user)
    BIND(URI(CONCAT("http://polimi.deib/hotrod#Customer", STR(?cid))) as ?customer)
    BIND(URI(CONCAT("http://polimi.deib/hotrod#Driver", MD5(STR(?did)))) as ?driver)
}
```

Listing 6.21 `CONSTRUCT` query for the HotR.O.D. stream in RSP-QL.

In this way, we can register the stream obtained to the engine to query it through the HotR.O.D. ontology. Listing 6.22 counts the number of times a *User* requests a car to a given *Customer*. Listing 6.23, instead, counts the number of times a *Driver* is assigned to a

given *User*.

```
PREFIX tr: <http://polimi.deib/tracing#>
PREFIX ht: <http://polimi.deib/hotrod#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

REGISTER RSTREAM <s1> AS
SELECT ?uid ?cid (COUNT(*) AS ?n_requests)
FROM NAMED WINDOW <win> ON <constructHotrod> [RANGE PT30S STEP PT30S]
WHERE  {
    WINDOW <win> {
        ?u ht:requestCarTo ?c ;
          ht:sessionID ?uid .
        ?c ht:customerID ?cid .
    }
}
GROUP BY ?uid ?cid
```

Listing 6.22 Count the number of times a *User* requests a car to a given *Customer* in RSP-QL.

```
PREFIX tr: <http://polimi.deib/tracing#>
PREFIX ht: <http://polimi.deib/hotrod#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

REGISTER RSTREAM <s1> AS
SELECT ?uid ?did (COUNT(*) AS ?n_requests)
FROM NAMED WINDOW <win> ON <constructHotrod> [RANGE PT30S STEP PT30S]
WHERE  {
    WINDOW <win> {
        ?u ht:assignedDriver ?d ;
          ht:sessionID ?uid .
        ?d ht:licensePlateNumber ?did .
    }
}
GROUP BY ?uid ?did
```

Listing 6.23 Count the number of times a *Driver* is assigned to a given *User* in RSP-QL.

Chapter 7

Conclusions and Future Work

In this thesis work, we framed and investigated the research challenges, around the problem of *observability* for software systems, that are getting industrial attention.

In Chapter 3, we described the issues around modern systems architectures motivating our work, and we clarified the boundaries of the *observability* problem. Finally, we formulated two research questions on how to *expose* (**OP1**) and how to *make sense* (**OP2**) of system behaviour at runtime. Last but not least, we elicited a set of requirements for the solution to satisfy.

In Chapter 4, we presented the design of our solution. In particular, to solve **OP1**, we presented a unifying data model based on the concept of *event* that reconciles metrics, logs and trace data. To solve **OP2** we showed how a stream processing approach fulfils requirements. Given precedent works applying stream processing successfully on metrics and logs, we focused on the specification for a Trace Stream Processor (TSP). Following the *Design Science* framework, we designed a prototype, *Kaiju*, together with *Rim*, a reproducible environment emulating the context addressed by *Kaiju*. Finally, we devised a stream reasoning approach to cope with both the problems presenting a specification for an RDF stream processing engine.

In Chapter 5, we described our implementation experience. We provided implementation details for *Kaiju* and *Rim*. Moreover, we proposed an ontology for trace data based on the OpenTracing ontology and we explained how we implement an RSP engine ingesting RDF trace data.

In Chapter 6, we compared *Kaiju* with respect to state-of-the-art distributed tracing tools, and to the typical use cases they address. We showed how *Kaiju* approaches the problem in different ways than state-of-the-art solutions, but can provide the same functionalities and deal with a broader set of use cases. We demonstrated all this in *Rim*. Moreover, we showed how *Kaiju* could be extended to process other types of *observations*

and to correlate heterogeneous data on system behaviour. Finally, we also provide an explorative analysis on the use for *observability* of an RSP engine.

7.1 Discussion

In this work, we introduced a more structured approach towards *observability* for software discussing it as a research problem. We proposed a definition focusing on the concept of *system behaviour*, and we exploited it to formulate better the *observability* problem. We identified two separated but tightly coupled sub-problems: **OP1** *How can we expose the system behaviour through the outputs?* and **OP2** *How can we make sense of system behaviour?*. We addressed these problems discussing two related research questions and eliciting the related requirements.

In **Q1** we asked if it is possible to unify data models and processing pipelines of metrics, logs and trace data (i.e., *observations*) to provide a single and significative output for the observable behaviour of a software system. For this question, we elicited requirements pointing out the dependency of *observations* on time and the necessity to provide both structural and semantic interoperability. Therefore, we proposed a data model based on the concept of event to establish time as a first-class citizen, and we offered a unique unified structure for *observations* through the definition of a *payload* and a *context*. This data model has several benefits:

i) it could enable a unique instrumentation API avoiding replication or split of information under multiple formats or with different semantics;

ii) the context enables the possibility to slice, aggregate and join *observability events*, with different types of payloads, over arbitrary shared dimensions;

iii) a single perspective can help in determining a more general set of guidelines to deal with the trade-off between collecting too many data and not exposing enough information on system behaviour.

However, we pointed out the importance of still considering metrics, logs and traces data as the three main abstractions to expose and make sense of system behaviour. Indeed, they define different payloads, and each of them has its specificity (calling for different guidelines to understand what should be exposed) and different processing mechanisms (must be taken into account to make sense of the system behaviour).

In **Q2** we asked if it is possible to make sense in near real-time of information needs about the system observable behaviour, considering the available *observations* at runtime,

and despite data heterogeneity. For this question, we elicited requirements pointing out the need for velocity and reactivity in processing *observations*, the importance of taking into account their timing relations and the need to handle the complexity of data. Given the elicited requirements we demonstrated how this problem belongs to the *Information Flow Processing* domain and, therefore, can be adequately addressed by a stream processing solution. Since the validity of a stream processing solution for metrics and logs has been already proved, we decided to focus on trace data. Indeed, some research works highlights the potentialities of dynamic analysis for trace data, but state-of-the-art distributed tracing systems, e.g. *Jaeger*, only focuses on static analysis. With the development and evaluation of *Kaiju*, we demonstrated how a Trace Stream Processor could cover almost any use case related to tracing and eventually postpone storing and static analysis.

Moreover, we showed how it could empower a posteriori sampling to store only interesting traces. We highlighted the necessity to process *observations* expressing both *transforming* and *detecting* rules, and for this reason, we selected *Esper* as a stream processing engine capable of providing both types of rules. We also showed how EPL, the language offered by Esper, can effectively deal with complex data also exploiting the contained event syntax. Furthermore, listeners offer a flexible mechanism to define the logic to handle outputs for each rule, and the API implemented allows to install and remove rules at runtime.

In our comparison of *Jaeger* and *Kaiju*, we highlighted the indexing overhead introduced from the former and the effectiveness of the latter in handling high volumes of data.

Jaeger enables a static analysis of traces through a visualisation allowing to inspect them. It mainly addresses the use cases related to the *diagnosis of steady-state problems* and partially the *workload modelling* offering a services dependencies graph.

As shown in our evaluation, *Kaiju* can partially cover this use cases even if both require a visualisation layer to be adequately addressed. Currently, an interface is not provided in *Kaiju*, even if, we showed we could provide a suitable API to implement a visualisation similar to the one of *Jaeger*. Moreover, *Kaiju* enables a dynamic analysis of traces through: (i) continuous querying, (ii) fire-and-forget queries on specific windows, and (iii) event-based pattern rules to detect complex events.

Therefore, thanks to the expressiveness of its language and the stream-oriented approach, *Kaiju* allows processing request-scoped *observations* covering all other use cases of tracing: *anomaly detection*, *distributed profiling*, *resource usage attribution*.

In this thesis, we also presented *Rim*, a reproducible environment that allows emulating a distributed system running a multi-node multiple-instance application instrumented to output request-scoped *observations*. In our evaluation, we showed how *Rim* could reproduce typical issues of the context reproduced allowing to check the effectiveness of tools and methodologies to observe the system status at runtime.

Finally, we tried to address both the challenges simultaneously. Although we could extend *Kaiju* to handle multiple types of data, this is not the prescribed way to handle heterogeneity. Alternatively, a stream reasoning approach as RDF Streaming Processing (RSP) can handle both variety and velocity. To this extent, we proposed an ontology for the OpenTracing specification, and we annotated traces in RDF enabling ingestion of trace RDF graphs through an RSP engine. We then provided an explorative analysis showing how it can solve a set of use cases presented for *Kaiju*, and how it allows querying also application level details.

7.2 Limitations and future work

The main limitation of our work is related to the fact we operated on an emulated environment. We do not have the opportunity to inspect traces produced by a real production cluster, and therefore, our analysis is limited to the set of issues reproducible in *Rim*. We plan, therefore, to extend our evaluation to an industrial scenario to better understand the advantages and limits of our approach, also with respect to real users' traffic.

Furthermore, we observed the lack of instruments capable of evaluating tools dealing with trace data. We evaluated *Kaiju* on the tracing use cases, but we devised the necessity of a shared benchmark for a more systematic comparison. Recent works, like the one from *Zhou et al.* [68] proposed an open source dataset for this purpose. However, their work misses the opportunity to release trace data compliant with the *OpenTracing* specification. A future work may be to extend *Rim* to produce a referential benchmark, or to become a tool for *technical action research* (TAR) as defined in [65], i.e. a tool for the validation of artifacts related to *observability* applied in a realistic case.

A second limitation consists in the fact that we do not consider and include in *Rim* an orchestration system. Nowadays clusters are mainly run by mean of these tools and, therefore, it is necessary to process also their outputs in order to obtain an overall view of systems.

Another aspect that should be investigated in greater details is the scalability of *Kaiju*. A shown in [8] it can be useful to distribute the stream processor, exploiting pre-processing at the node level, to reduce the overhead in the network and to process less data contem-

poraneously. Moreover, we implemented *Kaiju* as a *lambda* architecture [45] exploiting *Jaeger* to offer a parallel batch solution for data gathered. It would be useful to investigate further the possibility of a *kappa* architecture for trace data and, in general, for *observability*.

To conclude, concerning the proposed stream reasoning approach, it is necessary to extend the explorative analysis made with the RSP engine to RDF logs and metrics. For metrics, moreover, an ontology should be defined once a specification is released for the OpenMetrics standard. Furthermore, it is worth to investigate other forms of stream reasoning. E.g., those based on CEP like [62] and those based on metric temporal logic like [37] looks promising for time-aware analysis of *observations*.

References

[1] Lior Abraham, John Allen, Oleksandr Barykin, Vinayak Borkar, Bhuwan Chopra, Ciprian Gerea, Daniel Merl, Josh Metzler, David Reiss, Subbu Subramanian, et al. Scuba: diving into data at facebook. *Proceedings of the VLDB Endowment*, 6(11): 1057–1067, 2013.

[2] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proc. VLDB Endow.*, 8(12):1792–1803, August 2015. ISSN 2150-8097. doi: 10.14778/2824032.2824076. URL http://dx.doi.org/10.14778/2824032.2824076.

[3] Sara Alspaugh, Bei Di Chen, Jessica Lin, Archana Ganapathi, Marti A Hearst, and Randy H Katz. Analyzing log analysis: An empirical study of user log mining. In *LISA*, pages 53–68, 2014.

[4] Marc Andreessen. Why software is eating the world. *The Wall Street Journal*, 20(2011): C2, 2011.

[5] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The cql continuous query language: semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, 2006.

[6] Gianluca Arbezzano. Observability according to me. URL https://gianarb.it/blog/observability. Accessed September 2018.

[7] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '02, pages 1–16, New York, NY, USA, 2002. ACM. ISBN 1-58113-507-6. doi: 10.1145/543613.543615. URL http://doi.acm.org/10.1145/543613.543615.

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

[9] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. C-sparql: a continuous query language for rdf data streams. *Intl. J. Semantic Computing*, 4(01):3–25, 2010.

[10] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. Incremental reasoning on streams and rich background knowledge. In *Extended Semantic Web Conference*, pages 1–15. Springer, 2010.

[11] Peter Bourgon. Metrics, tracing and logging, . URL https://peter.bourgon.org/blog/2017/02/21/metrics-tracing-and-logging.html. Accessed September 2018.

[12] Peter Bourgon. Observability signals, . URL https://peter.bourgon.org/blog/2018/08/22/observability-signals.html. Accessed September 2018.

[13] Jean-Paul Calbimonte, Oscar Corcho, and Alasdair JG Gray. Enabling ontology-based access to streaming data sources. In *International Semantic Web Conference*, pages 96–111. Springer, 2010.

[14] Sharma Chakravarthy and Deepak Mishra. Snoop: An expressive event specification language for active databases. *Data & Knowledge Engineering*, 14(1):1–26, 1994.

[15] World Wide Web Consortium et al. RDF 1.1 Primer. 2014.

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

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

[18] Gianpaolo Cugola and Alessandro Margara. Tesla: A formally defined event specification language. In *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*, DEBS '10, pages 50–61, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-927-5. doi: 10.1145/1827418.1827427. URL http://doi.acm.org/10.1145/1827418.1827427.

[19] José C Cunha, João Lourenço, and Vítor Duarte. Debugging of parallel and distributed programs. *Parallel program development for cluster computing: methodology, tools and integrated environments*, pages 97–129, 2001.

[20] Datadog. *Monitoring in the cloud*. 2016. URL https://www.datadoghq.com/pdf/monitoring-in-the-cloud-ebook.pdf.

[21] Thomas H Davenport. Competing on analytics. *harvard business review*, 84(1):98, 2006.

[22] Emanuele Della Valle, Stefano Ceri, Frank Van Harmelen, and Dieter Fensel. It's a streaming world! reasoning upon rapidly changing information. *IEEE Intelligent Syst.*, (6):83–89, 2009.

[23] Emanuele Della Valle, Daniele Dell'Aglio, and Alessandro Margara. Taming velocity and variety simultaneously in big data with stream reasoning: Tutorial. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, DEBS '16, pages 394–401, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4021-2. doi: 10.1145/2933267.2933539. URL http://doi.acm.org/10.1145/2933267.2933539.

[24] Daniele Dell'Aglio, Emanuele Della Valle, Jean-Paul Calbimonte, and Oscar Corcho. Rsp-ql semantics: a unifying query model to explain heterogeneity of rdf stream processing systems. *Intl. J. on Semantic Web and Information Systems (IJSWIS)*, 10(4): 17–44, 2014.

[25] Daniele Dell'Aglio, Emanuele Della Valle, Frank van Harmelen, and Abraham Bernstein. Stream reasoning: A survey and outlook. *Data Science*, (Preprint):1–24.

[26] Clóvis Holanda do Nascimento, Rodrigo Elia Assad, Bernadette Farias Lóscio, and Silvio Romero Lemos Meira. Ontolog: A security log analyses tool using web semantic and ontology. *Web Application Security*, page 1, 2010.

[27] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. Microservices: yesterday, today, and tomorrow. In *Present and Ulterior Software Engineering*, pages 195–216. Springer, 2017.

[28] Patrick Th Eugster, Pascal A Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM computing surveys (CSUR)*, 35(2):114–131, 2003.

[29] John Fahl. So What is Observability Anyway. URL https://logz.io/blog/what-is-observability/. Accessed september 2018.

[30] Micheal Feathers. Microservices Until Macro Complexity. URL https://michaelfeathers.silvrback.com/microservices-until-macro-complexity. Accessed September 2018.

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

[32] Rodrigo Fonseca, Michael J Freedman, and George Porter. Experiences with tracing causality in networked services. *INM/WREN*, 10:10–10, 2010.

[33] Nicole Forsgren, J Humble, and G Kim. Accelerate: State of devops, 2018.

[34] Clinton Gormley and Zachary Tong. *Elasticsearch: The Definitive Guide: A Distributed Real-Time Search and Analytics Engine*. " O'Reilly Media, Inc.", 2015.

[35] OpenTracing Working Group. OpenTracing specification repository. URL https://github.com/opentracing/specification. Accessed January 2018.

[36] Thomas R Gruber. Toward principles for the design of ontologies used for knowledge sharing. *International journal of human-computer studies*, 43(5-6):907–928, 1995.

[37] Fredrik Heintz, Jonas Kvarnström, and Patrick Doherty. Stream reasoning in dyknow: A knowledge processing middleware system. In *Proc. 1st Intl Workshop Stream Reasoning*, 2009.

[38] Jonathan Kaldor, Jonathan Mace, Michał Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, et al. Canopy: An end-to-end performance tracing and analysis system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 34–50. ACM, 2017.

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

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

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

[42] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[43] Danh Le-Phuoc, Minh Dao-Tran, Josiane Xavier Parreira, and Manfred Hauswirth. A native and adaptive approach for unified processing of linked streams and linked data. In *International Semantic Web Conference*, pages 370–388. Springer, 2011.

[44] Jonathan Leavitt. End-to-end tracing models: Analysis and unification, 2014.

[45] Jimmy Lin. The lambda and the kappa. *IEEE Internet Computing*, 21(5):60–66, 2017.

[46] David Luckham. *The power of events*, volume 204. Addison-Wesley Reading, 2002.

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

[48] Charity Majors. Observability for emerging infra: What got you here won't get you there. URL https://www.youtube.com/watch?v=oGC8C9z7TN4. Accessed september 2018.

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

[50] Marco Miglierina, Marco Balduini, Narges Shahmandi Hoonejani, Elisabetta Di Nitto, and Danilo Ardagna. Exploiting stream reasoning to monitor multi-cloud applications. *OrdRing@ ISWC*, 1059:33–36, 2013.

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

[52] Piyush Nimbalkar, Varish Mulwad, Nikhil Puranik, Anupam Joshi, and Tim Finin. Semantic interpretation of structured log files. In *2016 IEEE 17th International Conference on Information Reuse and Integration (IRI)*, pages 549–555. IEEE, 2016.

[53] Torben Bach Pedersen and Christian S Jensen. Multidimensional data modeling for complex data. In *Data Engineering, 1999. Proceedings., 15th International Conference on*, pages 336–345. IEEE, 1999.

[54] Raja R Sambasivan, Rodrigo Fonseca, Ilari Shafer, and Gregory R Ganger. 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.

[55] Stepan Shevtsov, Mihaly Berekmeri, Danny Weyns, and Martina Maggio. Control-theoretical software adaptation: A systematic literature review. *IEEE Transactions on Software Engineering*, 44(8):784–810, 2018.

[56] Yuri Shkuro. Evolving Distributed Tracing at Uber Engineering. URL https://eng.uber.com/distributed-tracing/. Accessed January 2018.

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

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

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

[60] Michael Stonebraker, Uğur Çetintemel, and Stan Zdonik. The 8 requirements of real-time stream processing. *SIGMOD Rec.*, 34(4):42–47, December 2005. ISSN 0163-5808. doi: 10.1145/1107499.1107504. URL http://doi.acm.org/10.1145/1107499.1107504.

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

[62] Riccardo Tommasini, Pieter Bonte, Emanuele Della Valle, Erik Mannens, Filip De Turck, and Femke Ongenae. Towards ontology-based event processing. In *OWL: Experiences and Directions–Reasoner Evaluation*, pages 115–127. Springer, 2016.

[63] Chengwel Wang, Soila P Kavulya, Jiaqi Tan, Liting Hu, Mahendra Kutare, Mike Kasick, Karsten Schwan, Priya Narasimhan, and Rajeev Gandhi. Performance troubleshooting in data centers: an annotated bibliography. *ACM SIGOPS Operating Systems Review*, 47(3):50–62, 2013.

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

[65] Roel Wieringa and Ayşe Moralı. Technical action research as a validation method in information systems design science. In *International Conference on Design Science Research in Information Systems*, pages 220–238. Springer, 2012.

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

[67] Felix Wolf and Bernd Mohr. Earl—a programmable and extensible toolkit for analyzing event traces of message passing programs. In *International Conference on High-Performance Computing and Networking*, pages 503–512. Springer, 1999.

[68] Jingwen Zhou, Zhenbang Chen, Ji Wang, Zibin Zheng, and Michael R Lyu. A data set for user request trace-oriented monitoring and its applications. *IEEE Transactions on Services Computing*, 11(4):699–712, 2018.

Acknowledgements

Milano, 20 Dicembre 2018

Ringrazio il Professor Emanuele Della Valle, che mi ha permesso di svolgere questo lavoro di tesi estremamente formativo e interessante. Ringrazio Riccardo Tommasini, che mi ha seguito in questo percorso mostrandomi la sua passione nel fare ricerca; grazie per i numerosi consigli e per avermi sempre permesso di esprimere il mio punto di vista, grazie per non avere mai smesso di motivarmi e per aver reso la tesi un'opportunità per imparare a divertirsi *getting things done*.

Ringrazio la mia famiglia, che non smette mai di supportarmi e di credere in me; grazie per avermi dato la possibilità di intraprendere e portare a termine questo cammino. Grazie per l'aiuto indispensabile di ogni giorno, la pazienza nelle sessioni d'esame, e per il desiderio grande di condividere insieme le difficoltà e i successi. Ringrazio mamma, sempre attenta a non farmi mancare nulla e capace in ogni occasione di tira(r)mi-sù e farmi tornare il sorriso con la sua cucina. Ringrazio papà, perchè sotto l'ironia nasconde sempre soddisfazione ed emozioni sincere. Ringrazio Andrea, che mi sopporta ogni giorno senza mai stancarsi di mostrarmi il suo affetto.

Ringrazio tutti i parenti che mi sono stati vicini spronandomi a dare sempre il massimo. Un grazie particolare a zia Mara che mi è sempre stata accanto nonostante la distanza, e a zia Liliana, che ha seguito il mio cammino condividendo la gioia per i miei piccoli e grandi traguardi. Ringrazio le mie nonne, Nancy e Titta, che mi portano sempre nel cuore e non mi hanno mai fatto mai mancare un pensiero (e soprattutto una preghiera) per ogni esame affrontato.

Ringrazio Corna, Marco, Marzo, Ste, Tommy e Vitty, amici veri e sinceri che spero di non perdere mai di vista. Grazie per ogni toro seduto, birra e avventura trascorsa insieme. Forse smetteremo di essere universitari, ma sicuramente non smetteremo di essere immaturi.

Ringrazio tutti i miei amici e quelli dell'OSG, che non smettono mai di starmi vicino e farmi sentire il loro affetto.

Ringrazio i mitici Cazzillos con cui ho condiviso il mio cammino politecnico; dei buoni compagni di viaggio sono difficili da trovare ma trovare degli amici è anche più raro. In particolare ringrazio Moro e Dado, *quelli della padelletta*, che hanno dato senso a tante giornate politecniche grazie alle nostre discussioni, ai mille caffè insieme e alle celebrazioni a base di George.

Ringrazio Bea, che ha sempre avuto un sorriso per colorare i momenti in cui tutto sembrava grigio e un abbraccio per rendere indimenticabili i traguardi raggiunti. Grazie perchè non hai mai smesso di farmi sentire la tua vicinanza e il tuo supporto. Grazie per avermi insegnato il bello di condividere le proprie emozioni e di sognare insieme.

Mario