Article

Cloud-native Observability: The Many-faceted Benefits of Structured and Unified Logging - A Case Study

Nane Kratzke

Lübeck University of Applied Sciences; nane.kratzke@th-luebeck.de
Correspondence: nane.kratzke@th-luebeck.de

Abstract: Background: Cloud-native software systems often have a much more decentralized structure and many independently deployable and (horizontally) scalable components, making it more complicated to create a shared and consolidated picture of the overall decentralized system state. Today, observability is often understood as a triad of collecting and processing metrics, distributed tracing data, and logging. The result is often a complex observability system composed of three stovepipes whose data is difficult to correlate. **Objective:** This study analyzes whether these three historically emerged observability stovepipes of logs, metrics and distributed traces could be handled more integrated and with a more straightforward instrumentation approach. **Method:** This study applied an action research methodology used mainly in industry-academia collaboration and common in software engineering. The research design utilized iterative action research cycles, including one long-term use case. **Results:** This study presents a unified logging library for Python and a unified logging architecture that uses the structured logging approach. The evaluation shows that several thousand events per minute are easily processable. **Conclusion:** The results indicate that a unification of the current observability triad is possible without the necessity to develop utterly new toolchains.

Keywords: cloud-native; observability; cloud computing; logging; structured logging; logs; metrics; traces; distributed tracing; log aggregation; log forwarding; log consolidation

1. Introduction

A "crypto winter" basically means that the prices for so-called cryptocurrencies such as Bitcon, Ethereeum, Solana, etc. fell sharply on the crypto exchanges and then stay low. The signs were all around in 2022: the failure of the TerraUSD crypto project in May 2022 sent an icy blast through the market, then the cryptocurrency lending platform Celsius Network halted withdrawals, prompting a sell-off that pushed Bitcoin to a 17-month low.

This study logged such a "crypto winter" on Twitter more by accident than by intention. Twitter was simply selected as an appropriate use case to evaluate a unified logging solution for cloud-native systems and decided to log Tweets containing stock symbols like $USD or $EUR. It turned out that most symbols used on Twitter are not related to currencies like $USD (US-Dollar) or stocks like $AAPL (Apple) but to Cryptocurrencies like $BTC (Bitcoin) or $ETH (Ethereum). The Twitter community therefore seems to be quite cryptocurrency-savvy. So, although some data of this 2022 crypto winter will be presented in this paper, this paper will take more the methodical part into focus and will address how such and further data could be collected more systematically in distributed cloud-native applications. The paper will at least show that even complex observability of distributed systems can be reached, simply by logging events to stdout.

Observability measures how well a system's internal state can be inferred from knowledge of its external outputs. The concept of observability was initially introduced by the Hungarian-American engineer Rudolf E. Kálmán for linear dynamical systems [1,2]. However, observability also applies to information systems and is of particular interest to fine-grained and distributed cloud-native systems that come with a very own set of observability challenges.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 25 August 2022 doi:10.20944/preprints202208.0427.v1

Traditionally, the responsibility for observability is (was?) with operations (Ops). With the emergence of DevOps, we can observe a shift of Ops responsibilities to developers. So, observability is evolving more and more into a Dev responsibility. Observability should ideally already be considered during the application design phase and not be regarded as some "add-on" feature for later expansion stages of an application. The current discussion about observability began well before the advent of cloud-native technologies like Kubernetes. A widely cited blog post by Cory Watson from 2013 shows how engineers at Twitter looked for ways to monitor their systems as the company moved from a monolithic to a distributed architecture [3–5]. One of the ways Twitter did this was by developing a command-line tool that engineers could use to create their dashboards to keep track of the charts they were creating. While CI/CD tools and container technologies often bridge Dev and Ops in one direction, observability solutions close the loop in the opposite direction, from Ops to Dev [4]. Observability is thus the basis for data-driven software development (see Fig. 1 and [6]). As developments around cloud(-native) computing progressed, more and more engineers began to "live in their dashboards." They learned that it is not enough to collect and monitor data points but that it is necessary to address this problem more systematically.

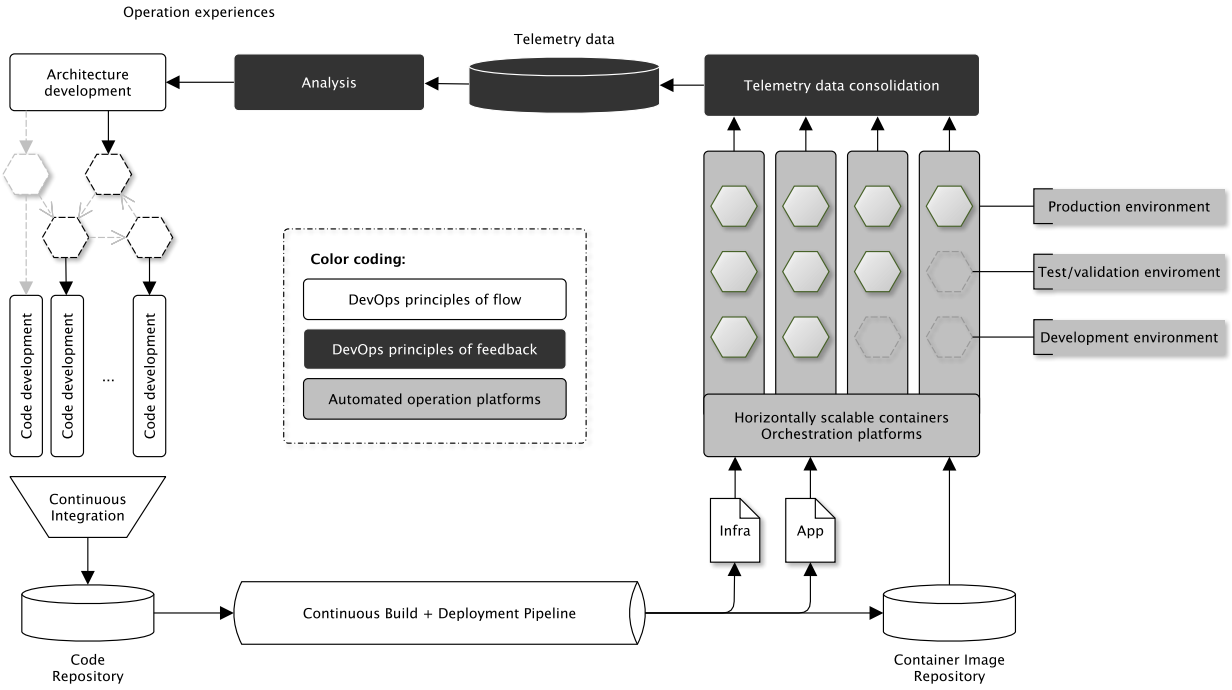

Figure 1. Observability can be seen as a feedback channel from Ops to Dev (adopted from [4] + [6]).

2. Problem description

Today, observability is often understood as a triad. Observability of distributed information systems is typically achieved through the collection and processing of metrics (quantitative data primarily as time-series), distributed tracing data (execution durations of complex system transactions that flow through services of a distributed system), and logging (qualitative data of discrete system events often associated with timestamps but encoded as unstructured strings). Consequently, three stacks of observability solutions have emerged, and the following somehow summarizes the current state of the art.

- **Metrics:** Here, quantitative data is often collected in time series, e.g., how many requests a system is currently processing. The metrics technology stack is often characterized by tools such as Prometheus and Grafana.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 25 August 2022 doi:10.20944/preprints202208.0427.v1

- **Distributed tracing** involves following the path of transactions along the components of a distributed system. The tracing technology stack is characterized by tools such as Zipkin or Jaeger, and the technologies are used to identify and optimize particularly slow or error-prone substeps of distributed transaction processing.
- **Logging** is probably as old as software development itself, and many developers, because of the log ubiquity, are unaware that logging should be seen as part of holistic observability. Logs are usually stored in so-called log files. Primarily qualitative events are logged (e.g. user XYZ logs in/out). An event is usually attached to a log file in a text line. Often the implicit and historically justifiable assumption prevails with developers that these log files are read and evaluated primarily by administrators (thus humans). However, that is hardly the case anymore. It is becoming increasingly common for the contents of these log files to be forwarded to a central database through "log forwarders" so that they can be evaluated and analyzed centrally. The technology stack is often characterized by tools such as Fluentd, FileBeat, LogStash for log forwarding, databases such as ElasticSearch, Cassandra or simply S3 and user interfaces such as Kibana.

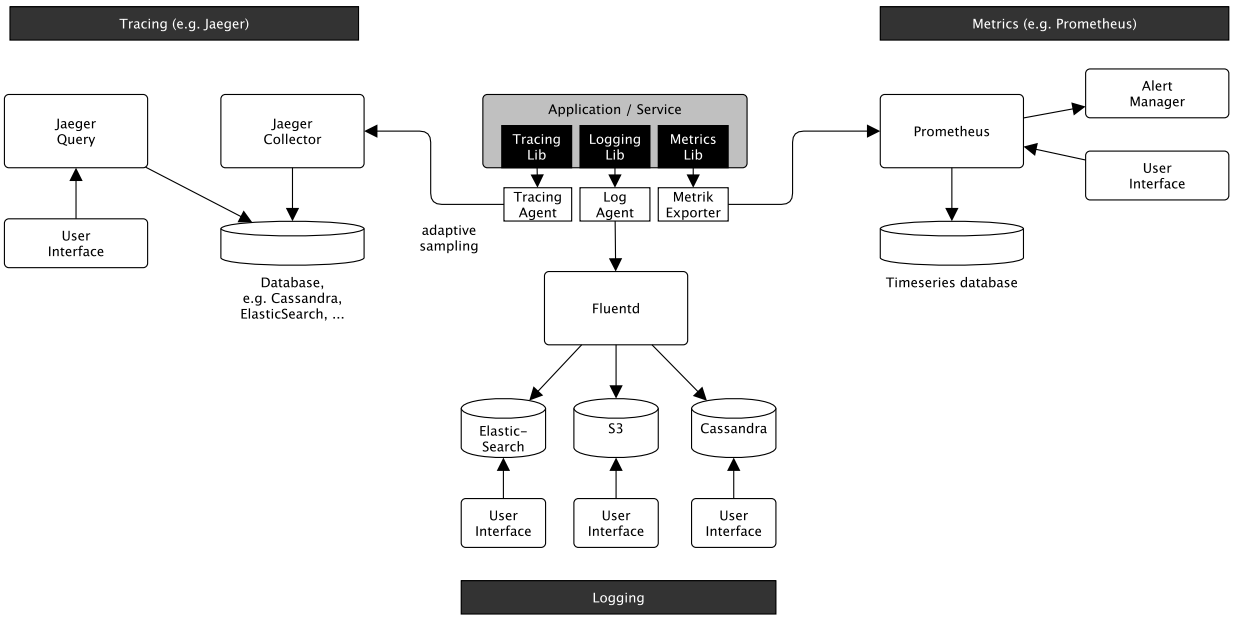

Figure 2. An application is quickly surrounded by a complex observability system when metrics, tracing and logs are captured with different observability stacks.

Incidentally, all three observability pillars have in common that software to be developed must be somehow instrumented. This instrumentation is normally done using programming language-specific libraries. Developers often regard distributed tracing instrumentation in particular as time-consuming. Also, which metric types (counter, gauge, histogram, history, and more) are to be used in metric observability solutions such as Prometheus often depends on Ops experience and is not always immediately apparent to developers. Certain observability hopes fail simply because of wrongly chosen metric types. Only system metrics such as CPU, memory, and storage utilization can be easily captured in a black-box manner (i.e., without instrumentation in the code). However, these data are often only of limited use for the functional assessment of systems. For example, CPU utilization provides little information about whether conversion rates in an online store are developing in the desired direction.

So, current observability solutions are often based on these three stovepipes for logs, metrics, and traces. The result is an application surrounded by a complex observability system whose isolated datasets can be difficult to correlate. Fig. 2 focuses on the application

 doi:10.20944/preprints202208.0427.v1

(i.e., the object to be monitored) and triggers the question, whether it is justified to use three complex subsystems and three types of instrumentation, which always means three times the instrumentation and data analysis effort of isolated data silos.

The often-used tool combination of ElasticSearch, LogStash, and Kibana is often used for logging and has even been given a catchy acronym: ELK-Stack [3]. The ELK stack can be used to collect metrics and using the plugin APM also for distributed tracing. So, at least for the ELK stack, the three stovepipes are not clearly separable or disjoint. The separateness is somewhat historically "suggested" than technologically given. Nevertheless, this tripartite division into metrics, tracing and logging is very formative for the industry, as shown, for example, by the OpenTelemetry project [7]. OpenTelemetry is currently in the incubation stage at the Cloud Native Computing Foundation and provides a collection of standardized tools, APIs, and SDKs to instrument, generate, collect, and export telemetry data (metrics, logs, and traces) to analyze the performance and behaviour of software systems. OpenTelemetry thus standardizes observability but hardly aims to overcome the columnar separation into metrics, tracing, and logging.

In past and current industrial action research [4,6,8–14], I came across various cloud-native applications and corresponding engineering methodologies like the 12-factor app (see 4.1) and learned that the discussion around observability is increasingly moving beyond these three stovepipes and taking a more nuanced and integrated view. There is a growing awareness of integrating and unifying these three pillars, and more emphasis is being placed on analytics.

The **research question** arises whether these three historically emerged observability stovepipes of logs, metrics and distributed traces could be handled more integrated and with a more straightforward instrumentation approach. The results of this action research study shows that this unification potential could be surprisingly easy to realize. This paper presents the methodology in Sec. 3 and its results in Sec. 4 (including a logging prototype in Sec 4.4 and its evaluation results in 4.5 as the **main contribution** of this paper to the field). The results are discussed in Sec. 5. Furthermore, the study presents related work in Sec. 6 and concludes its findings as well as future promising research directions in Sec. 7.

3. Methodology

This study followed the action research methodology as a proven and well-established research methodology model for industry-academia collaboration in the software engineering context to analyze the research-question mentioned above. Following the recommendations of Petersen et al. [15], a research design was defined that applied iterative action research cycles (see Fig. 3):

1. **Diagnosis** (Diagnosing according to [15])
2. **Prototyping** (Action planning, design and taking according to [15])
3. **Evaluation** including a may be required redesign (Evaluation according to [15])
4. **Transfer** learning outcomes to further use cases (Specifying learning according to [15])

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 25 August 2022 doi:10.20944/preprints202208.0427.v1

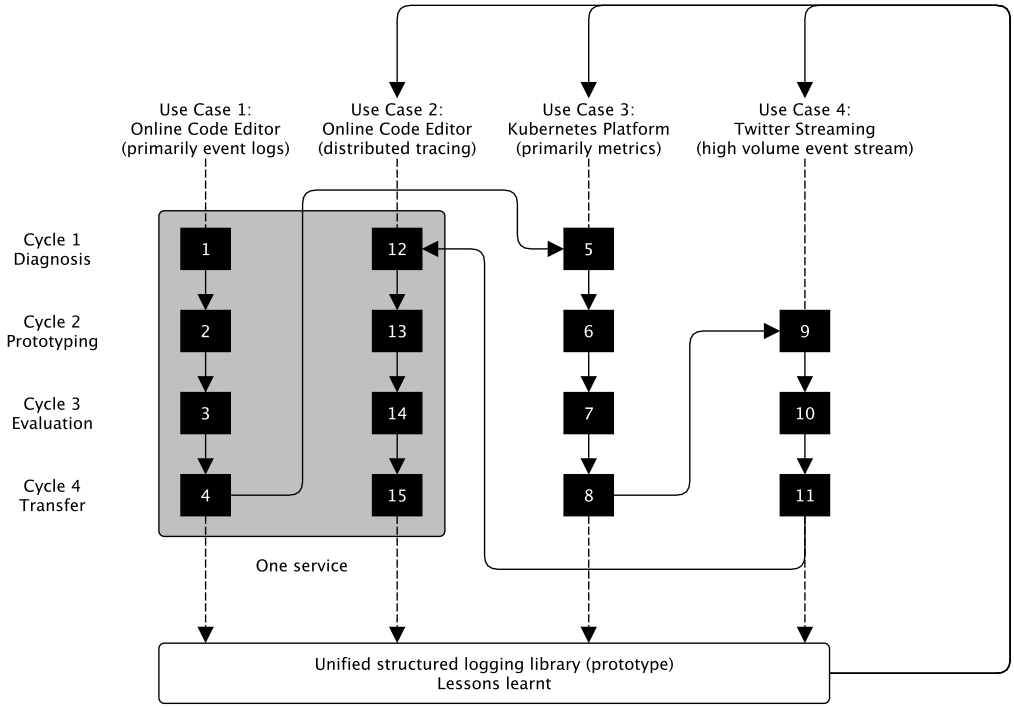

Figure 3. Action research methodology of this study

With each of the following use cases insights were transferred from the previous use case into a structured logging prototype (see Fig. 3). The following use cases have been studied and evaluated.

- **Use Case 1:** Observation of qualitative events occurring in an existing solution (online code editor; https://codepad.th-luebeck.dev, this use case was inspired by our research [11])
- **Use Case 2:** Observation of distributed events along distributed services (distributed tracing in an existing solution of an online code editor, see UC1)
- **Use Case 3:** Observation of quantitative data generated by a technical infrastructure (Kubernetes platform, this use case was inspired by our research [14])
- **Use Case 4:** Observation of a massive online event stream to gain experiences with high-volume event streams (we used Twitter as a data source and tracked worldwide occurrences of stock symbols, this use case was inspired by our research [16,17])

4. Results

The analysis of cloud-native methodologies like the 12-factor app [18] has shown that to build observability, one should take a more nuanced and integrated view to integrate and unify these three pillars of metrics, traces, and logs to enable more agile and convenient analytics in feedback information flow in DevOps cycles (see Fig. 1). Two aspects that gained momentum in cloud-native computing are of interest:

- Recommendations on how to handle log forwarding and log consolidaion in cloud-native applications
- Recommendations to apply structured logging

Because both aspects guided the implementation of the logging prototype deeply, they will be explained in more details providing the reader the necessary context.

4.1. Twelve-factor apps

The 12-factor app is a method [18] for building software-as-a-service applications that pay special attention to the dynamics of organic growth of an application over time,

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 25 August 2022 doi:10.20944/preprints202208.0427.v1

the dynamics of collaboration between developers working together on a codebase, and avoiding the cost of software erosion. At its core, 12 rules (factors) should be followed to develop well-operational and evolutionarily developable distributed applications. This methodology harmonizes very well with microservice architecture approaches [3] and cloud-native operating environments like Kubernetes [19], which is why the 12-factor methodology is becoming increasingly popular. Incidentally, the 12-factor methodology does not contain any factor explicitly referring to observability, certainly not in the triad of metrics, tracing and logging. However, factor XI recommends how to handle logging:

> *Logs are the stream of aggregated events sorted by time and summarized from the output streams of all running processes and supporting services. Logs are typically a text format with one event per line.*
>
> *[...]*
>
> *A twelve-factor app never cares about routing or storing its output stream. It should not attempt to write to or manage log files.* ***Instead, each running process writes its stream of events to stdout.*** *[...] On staging or production deploys, the streams of all processes are captured by the runtime environment, combined with all other streams of the app, and routed to one or more destinations for viewing or long-term archiving. These archiving destinations are neither visible nor configurable to the app - they are managed entirely from the runtime environment.*

4.2. From logging to structured logging

The logging instrumentation is quite simple for developers and works mainly programming language specific but basically according to the following principle illustrated in Python.

A logging library must often be imported, defining so-called log levels such as DEBUG, INFO, WARNING, ERROR, FATAL, and others. While the application is running, a log level is usually set via an environment variable, e.g. INFO. All log calls above this level are then written to a log file.

```
import logging
logging.basicConfig(filename="example.log", level=logging.DEBUG)
logging.debug("Performing␣user␣check")
user = "Nane␣Kratzke"
logging.info(f"User␣{␣user␣}␣tries␣to␣log␣in.")
logging.warning(f"User␣{␣user␣}␣not␣found')
logging.error(f"User { user } has been banned.")
```

For example, line 5 would create the following entry in a log file:

```
INFO 2022-01-27 16:17:58 - User Nane Kratzke tries to log in
```

In a 12-factor app, this logging would be configured so that events are written directly to Stdout (console). The runtime environment (e.g., Kubernetes with FileBeat service installed) then routes the log data to the appropriate database taking work away from the developer that they would otherwise have to invest in log processing. This type of logging is well supported across many programming languages and can be consolidated excellently with the ELK stack (or other observability stacks).

Logging (unlike distributed tracing and metrics collection) is often not even perceived as (complex) instrumentation by developers. Often it is done on their own initiative. However, one can systematize this instrumentation somewhat and extend it to so-called "structured logging". Again, the principle is straightforward. One simply does not log lines of text like

```
INFO 2022-01-27 16:17:58 - User Nane Kratzke tries to log in
```

but instead, the same information in a structured form, e.g. using JSON:

```
{"log␣level": "info", "timestamp": "2022-01-27␣16:17:58", "event": "Log␣in",
    "user": "Nane␣Kratzke", "result": "success"}
```

In both cases, the text is written to the console. In the second case, however, a structured text-based data format is used that is easier to evaluate. In the case of a typical logging statement

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 25 August 2022 doi:10.20944/preprints202208.0427.v1

like *"User Max Mustermann tries to log in"* the text must first be analyzed to determine the user. This text parsing is costly on a large scale and can also be very computationally intensive and complex if there is plenty of log data in a variety of formats (which is the common case in the real world).

However, in the case of structured logging, this information can be easily extracted from the JSON data field "user". In particular, more complex evaluations become much easier with structured logging as a result. However, the instrumentation does not become significantly more complex, especially since there are logging libraries for structured logging. The logging looks in the logging prototype **log12** of this study like this:

```
import log12
[...]
log12.error("Log in", user=user, result="Not found", reason="Banned")
```

The resulting log files are still readable for administrators and developers (even if a bit more unwieldy) but much better processable and analyzable by databases such as ElasticSearch. Quantitative metrics can also be recorded in this way. Structured logging can thus also be used for the recording of quantitative metrics.

```
import log12
[...]
log12.info("Open requests", requests=len(requests))
```

```
{ "event": "Open requests", "requests": 42 }
```

What is more, this structured logging approach can also be used to create tracings. In distributed tracing systems, a trace ID is created for each transaction that passes through a distributed system. The individual steps are so-called spans. These are also assigned an ID (span ID). The span ID is then linked to the trace ID, and the runtime is measured and logged. In this way, the time course of distributed transactions can be tracked along the components involved, and, for example, the duration of individual processing steps can be determined.

4.3. Resulting and simplified logging architecture

So, if the two principles to print logs simply to stdout and to log in a structured and text-based data format are applied consequently. The resulting observability system complexity thus reduces from Fig. 2 to Fig. 4 because all system components can collect log, metric, and trace information in the same style that can be routed seamlessly from an operation platform provided log forwarder (already existing technology) to a central analytical database.

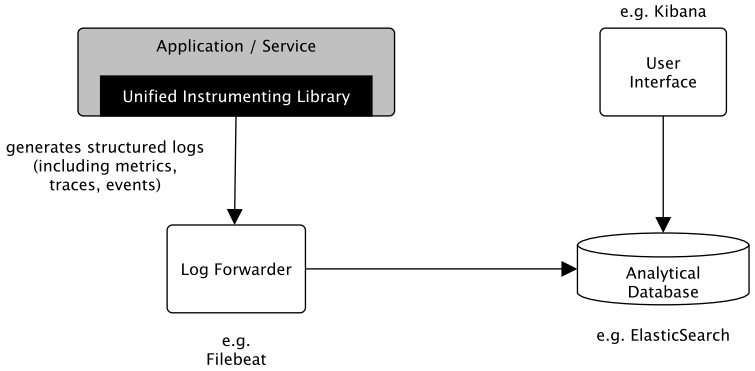

Figure 4. An observability system consistently based on structured logging with significantly reduced complexity.

4.4. Study outcome: Unified instrumentation via an structured logging library (prototype)

This paper will briefly explain below the way to capture events, metrics, and traces using the logging prototype that emerged. The prototype library log12 was developed in Python 3 but could implemented in other programming languages analogously.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 25 August 2022 doi:10.20944/preprints202208.0427.v1

log12 will create automatically for each event additional key-value attributes like an unique identifier (that is used to relate child events to parent events and even remote events in distributed tracing scenarios) and start and completion timestamps that can be used to measure the runtime of events (although known from distributed tracing libraries but not common for logging libraries). It is explained

- how to create a log stream,
- how an event in a log stream is created and logged,
- how a child event can be created and assigned to a parent event (to trace and record runtimes of more complex and dependent chains of events within the same process),
- and how to make use of the distributed tracing features to trace events that pass through a chain of services in a distributed service of services system).

The following lines of code create a log stream with the name "logstream" that is logged to stdout.

Listing 1: Creating an event log stream in log12

```
import log12
log = log12.logging("logstream",
    general="value", tag="foo", service_mark="test"
)
```

Each event and child events of this stream are assigned a set of key-value pairs:

- general="value"
- tag="foo"
- service_mark="test"

These log-stream-specific key-value pairs can be used to define selection criteria in analytical databases like ElasticSearch to filter events of a specific service only. The following lines of code demonstrate how to create a parent event and child events.

Listing 2: Event logging in log12 using blocks as structure

```
# Log events using the with clause
with log.event("Test", hello="World") as event:
    event.update(test="something")
    # adds event specific key value pairs to the event

    with event.child("Subevent 1 of Test") as ev:
        ev.update(foo="bar")
        ev.error("Catastrophe")
        # Explicit call of log (here on error level)

    with event.child("Subevent 2 of Test") as ev:
        ev.update(bar="foo")
        # Implicit call of ev.info("Success") (at block end)

    with event.child("Subevent 3 of Test") as ev:
        ev.update(bar="foo")
        # Implicit call of ev.info("Success") (at block end)
```

Furthermore, it is possible to log events in the event stream without the block style. That might be necessary for programming languages that do not support to close resources (here a log stream) at the end of a block. In this case programmers are responsible to close events using the **.info()**, **.warn()**, **.error()** log levels.

Listing 3: Event logging in log12 without blocks

```
# To log events without with-blocks is possible as well.
ev = log.event("Another test", foo="bar")
ev.update(bar="foo")
child = ev.child("Subevent of Another test", foo="bar")
ev.info("Finished")
# <= However, than you are are responsible to log events explicity
#    If parent events are logged all subsequent child events
#    are assumed to have closed successfully as well
```

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 25 August 2022 doi:10.20944/preprints202208.0427.v1

Using this type of logging to forward events along HTTP-based requests is also possible. This usage of HTTP-Headers is the usual method in distributed tracing. Two main capabilities are required for this [20]. First, extracting header information received by an HTTP service process must be possible. Secondly, it must be possible to inject the tracing information in follow-up upstream HTTP requests (in particular, the trace ID and span ID of the process initiating the request).

Listing 4 shows how **log12** supports this with an extract attribute at event creation and an inject method of the event that extracts relevant key-value pairs from the event so that they can be passed as header information along an HTTP request.

Listing 4: Extraction and injection of tracing headers in log12

```
import log12
import requests            # To generate HTTP requests
from flask import request  # To demonstrate Header extraction

with log.event("Distributed tracing", extract=request.headers) as ev:

    # Here is how to pass tracing information along remote calls
    with ev.child("Task 1") as event:
        response = requests.get(
            "https://qr.mylab.th-luebeck.dev/route?url=https://google.com",
            headers=event.inject()
        )
        event.update(length=len(response.text), status=response.status_code)
```

4.5. Evaluation of logging prototype in the definded use cases

Use Cases 1 and 2: Codepad is an online coding tool to share quickly short code snippets in online and offline teaching scenarios. It has been introduced during the Corona Pandemic shutdowns to share short code snippets mainly in online educational settings for 1st or 2nd semester computer science students. Meanwhile the tool is used in presence lectures and labs as well. The reader is welcome to try out the tool at https://codepad.th-luebeck.dev. This study used the Codepad tool in its **steps 1, 2, 3, and 4** of its action research methodology as an instrumentation use case (see Fig. 3) to evaluate the instrumentation of qualitative system events according to Sec. 4.4. Fig. 5 shows the Web-UI on the left and the resulting dashboard on the right. In a transfer step (**steps 12, 13, 14, and 15** of the action research methodolgy, see Fig. 3) the same product was used to evaluate distributed tracing instrumentation (not covered in detail by this report).

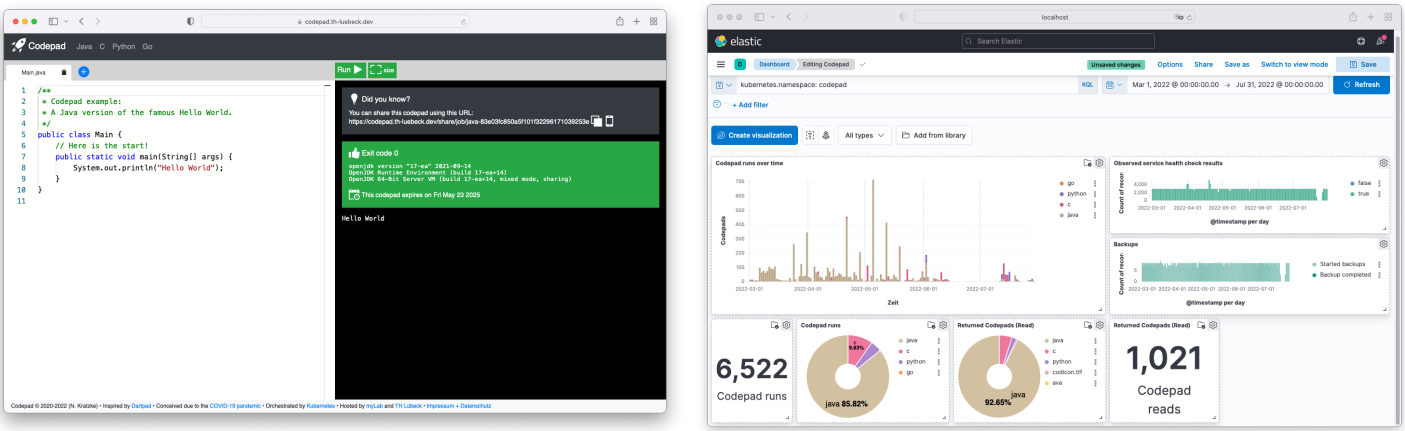

Figure 5. Use Cases 1 and 2: Codepad is an online coding tool to share quickly short code snippets in online and offline teaching scenarios. On the left the Web-UI. On the right the Kibana Dashboard used for observability in this study. Codepad was used as an instrumentation object of investigation.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 25 August 2022 doi:10.20944/preprints202208.0427.v1

The **Use Case 3 (steps 5, 6, 7, 8 of research methodology; Fig. 3)** observed an institutes infrastructure, the so-called myLab infrastructure. myLab (https://mylab.th-luebeck.dev) is a virtual laboratory that can be used by students and faculty staff to develop and host web applications. This use case was chosen to demonstrate that it is possible to collect primarily metrics based data over a long term using the same approach as in Use Case 1. A pod tracked mainly the resource consumption of various differing workloads deployed by more than 70 student web projects of different university courses. To observe this resource consumption the pod simply run periodically

- `kubectl top nodes`
- `kubectl top pods -all-namespaces`

against the cluster. This observation pod parsed the output of both shell commands and printed the parsed results in the structured logging approach presented in Sec. 4.4. Fig. 6 shows the resulting Kibana dashboard for demonstration purposes.

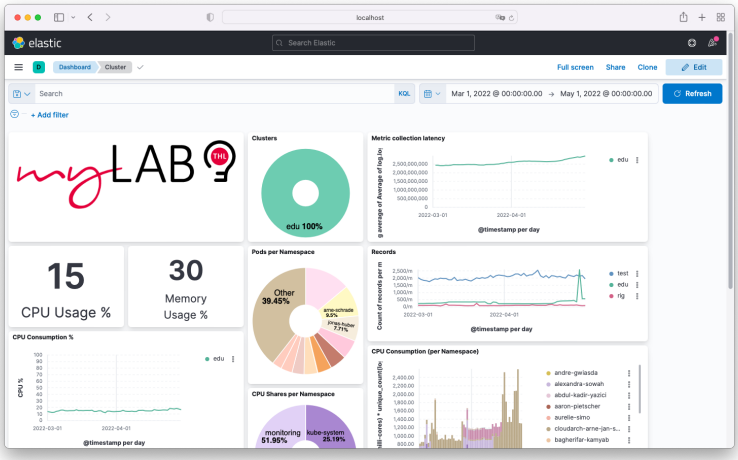

Figure 6. Use Case 3: The dashboard of the Kubernetes infrastructure under observation (myLab)

The **Use Case 4 (steps 9, 10, 11 of research methodology; Fig. 3)** left our own ecosystem and observed the public Twitter Event stream as a type representative for a high-volume and long-term observation of an external system. So, a system that was intentionally not under the direct administrative control of the study investigators. The Use Case 4 was designed as two phase study: The first screening phase was designed to gain experiences in logging high volume event streams and to provide necessary features and performance optimizations to the structured logging library prototype. The screening phase was designed to screen the complete and representative Twitter traffic as a kind of "ground truth". We were interested in the distribution of languages and stock symbols in relation to the general Twitter "background noise". This screening phase lasted from 20/01/2022 to 02/02/2022 and identified most used stock symbols. A long-term recording was then done as a second long-term evaluation phase and was used to track and record the most frequent used stock symbols identified in the screening phase. This evaluation phase lasted from Feb. 2022 until mid of August 2022. In this evaluation phase just one infrastructure downtime occurred due to a shutdown of electricity of the author's institute. However, this downtime was not due to or related to the presented unified logging stack (see Fig. 9).

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 25 August 2022 doi:10.20944/preprints202208.0427.v1

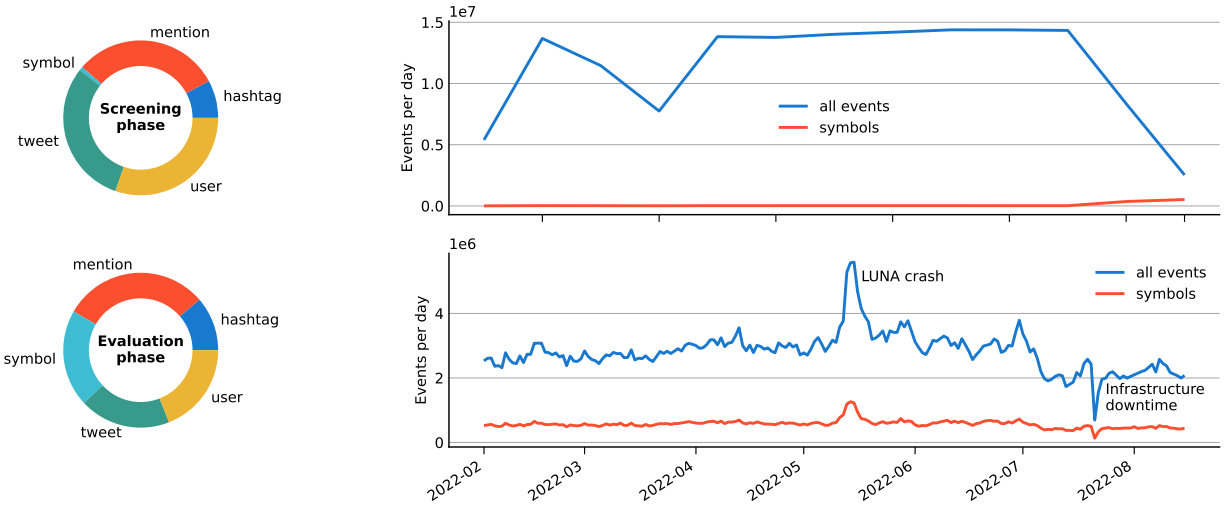

Figure 7. Recorded events (screening and evaluation phase of Use Case 4).

The recording was done using the following source code, compiled into a Docker container, that has been executed on a Kubernetes cluster that has been logged in **Use Case 1, 2, and 3**. FileBeat was used as a log forwarding component to a background ElasticSearch database. The resulting event log has been analyzed and visualized using Kibana. Kibana was used as well to collect the data in form of CSV-Files for the screening and the evaluation phase. The Fig. 7, 8, and 9 have been compiled from that data. This setting followed exactly the unified and simplified logging architecture presented in Fig. 4.

Listing 5: The used logging program to record Twitter stock symbols from the public Twitter Stream API

```
import log12, tweepy, os

KEY = os.environ.get("CONSUMER_KEY")
SECRET = os.environ.get("CONSUMER_SECRET")
TOKEN = os.environ.get("ACCESS_TOKEN")
TOKEN_SECRET = os.environ.get("ACCESS_TOKEN_SECRET")

LANGUAGES = [l.strip() for l in os.environ.get("LANGUAGES", "").split(",")]
TRACK = [t.strip() for t in os.environ.get("TRACKS").split(",")]

log = log12.logging("twitter stream")

class Twista(tweepy.Stream):

    def on_status(self, status):
        with log.event("tweet", tweet_id=status.id_str,
            user_id=status.user.id_str, lang=status.lang
        ) as event:
            kind = "status"
            kind = "reply" if status._json['in_reply_to_status_id'] else kind
            kind = "retweet" if 'retweeted_status' in status._json else kind
            kind = "quote" if 'quoted_status' in status._json else kind
            event.update(lang=status.lang, kind=kind, message=status.text)

            with event.child('user') as usr:
                name = status.user.name if status.user.name else "unknown"
                usr.update(lang=status.lang, id=status.user.id_str,
                    name=name,
                    screen_name=f"@{status.user.screen_name}",
                    message=status.text,
                    kind=kind
                )

```

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 25 August 2022 doi:10.20944/preprints202208.0427.v1


```
            for tag in status.entities['hashtags']:
                with event.child('hashtag') as hashtag:
                    hashtag.update(lang=status.lang,
                        tag=f"#{tag['text'].lower()}",
                        message=status.text,
                        kind=kind
                    )

            for sym in status.entities['symbols']:
                with event.child('symbol') as symbol:
                    symbol.update(lang=status.lang,
                        symbol=f"${sym['text'].upper()}",
                        message=status.text,
                        kind=kind
                    )
                    symbol.update(screen_name=f"@{status.user.screen_name}")

            for user_mention in status.entities['user_mentions']:
                with event.child('mention') as mention:
                    mention.update(lang=status.lang,
                        screen_name=f"@{user_mention['screen_name']}",
                        message=status.text,
                        kind=kind
                    )

record = Twista(KEY, SECRET, TOKEN, TOKEN_SECRET)
if LANGUAGES:
    record.filter(track=TRACK, languages=LANGUAGES)
else:
    record.filter(track=TRACK)
```

According to Fig. 7, just every 100th observed event in the screening phase was a stock symbol. That is simply the "ground-truth" on Twitter. If one is observing the public Twitter stream without any filter, that is what you get. So, the second evaluation phase recorded a very specific "filter bubble" of the Twitter stream. The reader should be aware, that the data presented in the following is a clear bias and not a representative Twitter event stream, it is clearly a stock market focused subset or to be even more precise: a cryptcocurrency focused subset, because almost all stock symbols on Twitter are related to cryptocurrencies.

It is possible to visualize the resulting effects using the recorded data. Fig. 8 shows the difference in language distributions of the screening phase (unfiltered ground-truth) and the evaluation phase (activated symbol filter). While in the screening phase English (en), Spanish (es), Portugese (pt), and Turkish (tr) are responsible for more than 3/4 of all traffic, in the evaluation phase almost all recorded Tweets are in English. So, on Twitter, the most stock symbol related language is clearly English.

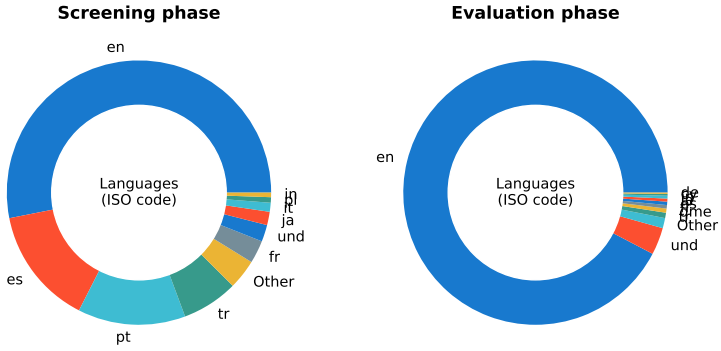

Figure 8. Observed languages (screening and evaluation phase of Use Case 4).

Although the cryptocurrency logging was used mainly as a use case for technical evaluation purposes of the logging library prototype, some interesting insights could be gained. E.g.,

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 25 August 2022 doi:10.20944/preprints202208.0427.v1

although Bitcoin (BTC) is likely the most prominent cryptocurrency, it is by far not the most frequent used stock symbol on Twitter. The most prominent stock symbols on Twitter are:

- ETH: Ethereum cryptocurrency
- SOL: Solana cryptocurrency
- BTC: Bitcoin cryptocurrency
- LUNA: Terra Luna cryptocurrency (replaced by a new version after the crash in May 2022)
- BNB: Binance Coin cryptocurrency

What is more, we can see interesting details in trends (see Fig. 9).

- The ETH usage on Twitter seems to reducing throughout our observed period.
- The SOL usage is on the opposite increasing, although we observed a sharp decline in July.
- The LUNA usage has a clear peak that correlates with the LUNA cryptocurrency crash in the mid of May 2022 (this crash was heavily reflected in the investor media).

The Twitter usage was not correlated with the curreny rates on crpytocurrency stock markets. However, changes in usage patterns of stock market symbols might be of interest for cryptocurrency investors as interesting indicators to observe. As this study shows, these changes can be easily tracked using structured logging approaches. Of course, this can be transferred to other social media streaming or general event streaming use cases like IoT (Internet of Things) as well.

5. Discussion

This style of a unified and structured observability was successfully evaluated on several use cases that made usage of a FileBeat/ElasticSearch-based observability stack. However, other observability stacks that can forward and parse structured text in a JSON-format will likely show the same results. The evaluation included a long-term test over more than six months for a high-volume evaluation use-case.

- On the one hand, it could be proven that such a type of logging can easily be used to perform classic metrics collections. For this purpose, BlackBox metrics such as CPU, memory, and storage for the infrastructure (nodes) but also the "payload" (pods) were successfully collected and evaluated in several Kubernetes clusters (see Fig. 6).
- Second, a high-volume use case was investigated and analyzed in-depth. Here, all English-language tweets on the public Twitter stream were logged. About 1 million events per hour were logged over a week and forwarded to an ElasticSearch database using the log forwarder FileBeat. Most systems will generate far fewer events (see Figure 7).

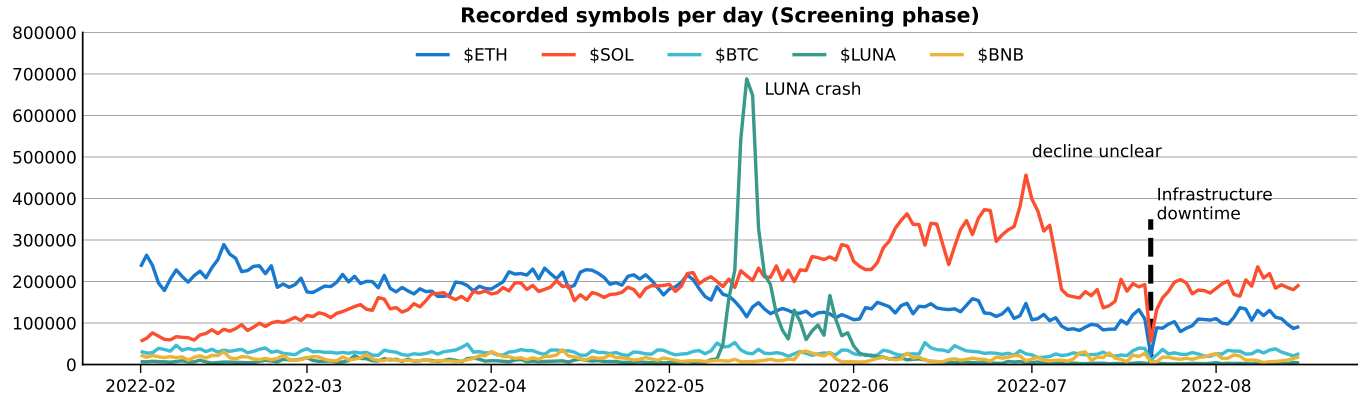

Figure 9. Recorded symbols per day (evaluation phase of Use Case 4).

 doi:10.20944/preprints202208.0427.v1

- In addition, the prototype logging library log12 is meanwhile used in several internal systems, including web-based development environments, QR code services, and e-learning systems, to record access frequencies to learning content, and to study learning behaviours of students.

5.1. Lessons learned

All use cases have shown that structured logging is easy to instrument and harmonizes well with existing observability stacks (esp. Kubernetes, Filebeat, ElasticSearch, Kibana). However, some aspects should be considered:

1. It is essential to apply structured logging, cause this can be used to log events, metrics, and traces in the same style.
2. Very often, only error-prone situations are logged. However, if you want to act in the sense of DevOps-compliant observability, you should also log normal - completely regular - behaviour. DevOps engineers can gain many insights from how normal users use systems in standard situations. So, the log level should be set to INFO, and not WARNING, ERROR, or above.
3. Cloud-native system components should rely on the log forwarding and log aggregation of the runtime environment. Never implement this on your own. You will double logic and end up with complex and may be incompatible log aggregation systems.
4. To simplify analysis for engineers, one should push key-value pairs of parent events down to child events. This logging approach simplifies analysis in centralized log analysis solutions - it simply reduces the need to derive event contexts that might be difficult to deduce in JSON document stores. However, this comes with the cost of more extensive log storage.
5. Do not collect aggregated metrics data. The aggregation (mean, median, percentile, standard deviations, sum, count, and more) can be done much more convenient in the analytical database. The instrumentation should focus on recording metrics data in a point-on-time style. According to our developer experience, developers are glad to be authorized to log only such simple metrics, especially when there is not much background knowledge in statistics.

5.2. Threats of validity and to be considered limitations of the study design

Action research is prone to drawing incorrect or non-generalizable conclusions. Logically, the significance is consistently highest within the considered use cases. In order to draw generalizable conclusions, this study defined use cases in such a way that intentionally different classes of telemetry data (logs, metrics, traces) were considered. It should be noted that the study design primarily considered logs and metrics but traces only marginally. Traces were not wholly neglected, however, but were analyzed less intensively.

The long-term acquisition was performed with a high-volume use case to cover certain stress test aspects. However, the reader must be aware, that the screening phase generated significantly higher data volumes in Use Case 4 than the evaluation phase. Therefore, to use stress test data from this study, one should look at the event volume of the screening phase of Use Case 4. Here, about ten thousand events per minute were logged for more than a week giving an impression of the performance of the proposed approach. The study data shows that the saturation limit should be far beyond these ten thousand events per minute. However, the study design did not pushed the system to its event recording saturation limits.

What is more, this study should not be used to derive any cryptocurrency related conclusions. Although some interesting aspects from Use Case 4 could be of interest for cryptocurency trading indicator generation. However, no detailed analysis on correlations between stock prices and usage frequencies of stock symbols on Twitter have been done.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 25 August 2022 doi:10.20944/preprints202208.0427.v1

6. Related work

There are relatively few studies dealing with observability as a main object of investigation in an academic understanding. The field is currently treated somewhat stepmotherly. However, an interesting and recent overview is provided by the survey of Usman et al. [21]. This survey provides a list of microservice-focused managed and unified observability services (Dynatrace, Datadog, New Relic, Sumo Logic, Solar Winds, Honeycomb). The presented research prototype of this study heads into the same direction, but tries to pursue the problem primarily on the instrumenting side using a more lightweight and unified approach. So, to address the client-side of the problem is obviously harder economical exploitable which is why the industry might address the problem preferable on the managed service side.

Of logs, metrics and distributed traces, distributed tracing is still considered in the most detail. In particular, the papers around Dapper [20] should be mentioned here, which had a significant impact on this field. A black box approach without instrumenting needs for distributed tracing is presented by [22]. This study, however, has seen tracing as only one of three aspects of observability and therefore follows a broader approach. A more recent review on current challenges and approaches of distributed tracing is presented by Bento et. al. [23].

6.1. Existing instrumenting libraries and observability solutions

Although the academic coverage of the observability field is expandable, in practice, there is an extensive set of existing solutions, especially for time series analysis and instrumentation. A complete listing is beyond the scope of this paper. However, from the disproportion of the number of academic papers to the number of real existing solutions, one quickly recognizes the practical relevance of the topic. Table 1 contains a list of existing database products often used for telemetry data consolidation to give the reader an overview without claiming completeness. This study used ElasticSearch as an analytical database.

Table 1. Often seen databases for telemetry data consolidation. Products used in this study are marked **bold** ⊗. Without claiming completeness.

Product	Organization	License	often seen scope
APM	Elastic	Apache 2.0	Tracing (add-on to ElasticSearch database)
ElasticSearch ⊗	Elastic	Apache/Elastic License 2.0	Logs, Tracing, (rarely Metrics)
InfluxDB	Influxdata	MIT	Metrics
Jaeger	Linux Foundation	Apache 2.0	Tracing
OpenSearch	Amazon Web Services	Apache 2.0	Logs, Tracing, (rarely Metrics); fork from ElasticSearch
Prometheus	Linux Foundation	Apache 2.0	Metrics
Zipkin	OpenZipkin	Apache 2.0	Tracing

Table 2 lists several frequently used forwarding solutions that developers can use to forward data from the point of capture to the databases listed in Table 1. In the context of this study, FileBeat was used as a log forwarding solution. It could be prooved that this solution is also capable to forward traces and metrics if applied in a structured logging setting.

Table 2. Often seen forwarding solutions for log consolidation. Products used in this study are marked **bold** ⊗. Without claiming completeness.

Product	Organization	License
Fluentd	FluentD Project	Apache 2.0
Flume	Apache	Apache 2.0
LogStash	Elastic	Apache 2.0
FileBeat ⊗	Elastic	Apache/Elastic License 2.0
Rsyslog	Adiscon	GPL
syslog-ng	One Identity	GPL

 doi:10.20944/preprints202208.0427.v1

An undoubtedly incomplete overview of instrumentation libraries for different products and languages is given in Table 3, presumably because each programming language comes with its own form of logging in the shape of specific libraries. To avoid this language-binding is hardly possible in the instrumentation context unless one pursues "esoteric approaches" like [22]. The logging library prototype is strongly influenced by the Python standard logging library but also by structlog for structured logging but without actually using these libraries.

Table 3. Often seen instrumenting libraries. Products that inspired the research prototype are marked **bold** ⊗. Without claiming completeness.

Product	Use Case	Organization	License	Remark
APM Agents ⊗	Tracing	Elastic	BSD 3	
Jaeger Clients	Tracing	Linux Foundation	Apache 2.0	
log	Logging	Go Standard Library	BSD 3	Logging for Go
log4j	Logging	Apache	Apache 2.0	Logging for Java
logging ⊗	Logging	Python Standard Library	GPL compatible	Logging for Python
Micrometer	Metrics	Pivotal	Apache 2.0	
OpenTracing	Tracing	OpenTracing	Apache 2.0	
prometheus	Metrics	Linux Foundation	Apache 2.0	
Splunk APM	Tracing	Splunk	Apache 2.0	
structlog ⊗	Logging	Hynek Schlawack	Apache 2.0, MIT	structured logging for Python
winston	Logging	Charlie Robbins	MIT	Logging for node.js

6.2. Standards

There are hardly any observability standards. However, a noteworthy standardization approach is the OpenTelemetry Specification [7] of the Cloud Native Computing Foundation [24], that tries to standardize the way of instrumentation. This approach corresponds to the core idea, which this study also follows. Nevertheless, the standard is still divided into Logs [25], Metrics [26] and Traces [27], which means that the conceptual triad of observability is not questioned. On the other hand, approaches like the OpenTelemetry Operator [28] for Kubernetes enable to inject auto-instrumentation libraries for Java, Node.js and Python into Kubernetes operated applications which is a feature that is currently not addressed by the present study. However, so-called service meshes also use auto-instrumentation. A developing standard here is the so-called Service Mesh Interface (SMI) [29].

7. Conclusions and Future Research Directions

Cloud-native software systems often have a much more decentralized structure and many independently deployable and (horizontally) scalable components, making it more complicated to create a shared and consolidated picture of the overall decentralized system state. Today, observability is often understood as a triad of collecting and processing metrics, distributed tracing data, and logging. But why except for historical reasons?

This study presents a unified logging library for Python [30] and a unified logging architecture (see Fig. 4) that uses a structured logging approach. The evaluation of four use cases shows that several thousand events per minute are easily processable and can be used to handle logs, traces, and metrics the same. At least, this study was able with a straight-forward approach to log the world-wide Twitter event stream of stock market symbols over a period of six months without any noteworthy problems. As a side effect, some interesting aspects how crypto-currencies are reflected on Twitter could be derived. This might be of minor relevance for this study but shows the overall potential of an unified and structured logging based observability approach.

The presented approach relies on an easy-to-use programming language-specific logging library that follows the structured logging approach. The long-term observation results of more than six months indicate that a unification of the current observability

triad of logs, metrics, and traces is possible without the necessity to develop utterly new toolchains. The trick is to

- use structured logging and
- apply log forwarding to a central analytical database
- in a systematic infrastructure- or platform-provided way.

Further research should therefore be concentrated on the instrumenting and less on the log forwarding and consolidation layer. If we instrument logs, traces, and metrics in the same style using the same log forwarding, we automatically generate correlatable data in a single data source of truth and we simplify analysis.

So, the observability road ahead may have several paths. On the one hand, we should standardize the logging libraries in a structured style like log12 in this study or the OpenTelemetry project in the "wild". Logging libraries should be comparably implemented in different programming languages and shall generate the same structured logging data. So, we have to standardize the logging SDKs and the data format. Both should be designed to cover logs, metrics, and distributed traces in a structured format. To simplify instrumentation further, we should additionally think about auto-instrumentation approaches, for instance, proposed by the OpenTelemetry Kubernetes Operator [28] and several Service Meshes like Istio [31] and corresponding standards like SMI [29].

Funding: This research received no external funding.

Data Availability Statement: The resulting research prototype of the developed structured logging library **log12** can be accessed here [30]. However, the reader should be aware, that this is prototyping software in progress.

Conflicts of Interest: The author declares no conflict of interest.

References

1. Kalman, R. On the general theory of control systems. *IFAC Proceedings Volumes* **1960**, *1*, 491–502. 1st International IFAC Congress on Automatic and Remote Control, Moscow, USSR, 1960, https://doi.org/https://doi.org/10.1016/S1474-6670(17)70094-8.
2. Kalman, R.E. Mathematical Description of Linear Dynamical Systems. *Journal of the Society for Industrial and Applied Mathematics Series A Control* **1963**, *1*, 152–192. https://doi.org/10.1137/0301010.
3. Newman, S. *Building Microservices*, 1st ed.; O'Reilly Media, Inc., 2015.
4. Kim, G.; Humble, J.; Debois, P.; Willis, J.; Forsgren, N. *The DevOps handbook: How to create world-class agility, reliability, & security in technology organizations*; IT Revolution, 2016.
5. Davis, C. *Cloud Native Patterns: Designing change-tolerant software*; Simon and Schuster, 2019.
6. Kratzke, N. *Cloud-native Computing: Software Engineering von Diensten und Applikationen für die Cloud*; Carl Hanser Verlag GmbH Co. KG, 2021.
7. The OpenTelemetry Authors. The OpenTelemetry Specification, 2021.
8. Kratzke, N.; Peinl, R. ClouNS - a Cloud-Native Application Reference Model for Enterprise Architects. In Proceedings of the 2016 IEEE 20th International Enterprise Distributed Object Computing Workshop (EDOCW), 2016, pp. 1–10. https://doi.org/10.1109/EDOCW.2016.7584353.
9. Kratzke, N.; Quint, P.C. Understanding Cloud-native Applications after 10 Years of Cloud Computing - A Systematic Mapping Study. *Journal of Systems and Software* **2017**, *126*, 1–16. https://doi.org/10.1016/j.jss.2017.01.001.
10. Kratzke, N. A Brief History of Cloud Application Architectures. *Applied Sciences* **2018**, *8*. https://doi.org/10.3390/app8081368.
11. Kratzke, N. How programming students trick and what JEdUnit can do against it. In *Computer Supported Education* ; Lane, H.C.; Zvacek, S.; Uhomoibhi, J., Eds.; Springer International Publishing , 2020; pp. 1–25. CSEDU 2019 - Revised Selected Best Papers (CCIS), https://doi.org/10.1007/978-3-030-58459-7_1.
12. Kratzke, N. Einfachere Observability durch strukturiertes Logging. *Informatik Aktuell* **2022**.
13. Kratzke, N.; Siegfried, R. Towards Cloud-native Simulations - Lessons learned from the front-line of cloud computing. *Journal of Defense Modeling and Simulation* **2020**. https://doi.org/10.1177/1548512919895327.
14. Truyen, Eddy.; Kratzke, Nane.; Van Landyut, Dimitri.; Lagaisse, Bert.; Joosen, Wouter. Managing Feature Compatibility in Kubernetes: Vendor Comparison and Analysis. *IEEE Access* **2020**, *8*, "228420–228439". https://doi.org/10.1109/ACCESS.2020.3045768.
15. Petersen, K.; Gencel, C.; Asghari, N.; Baca, D.; Betz, S. Action Research as a Model for Industry-Academia Collaboration in the Software Engineering Context. In Proceedings of the Proceedings of the 2014 International Workshop on Long-Term Industrial Collaboration on Software Engineering; Association for Computing Machinery: New York, NY, USA, 2014; WISE '14, p. 55–62. https://doi.org/10.1145/2647648.2647656.

16. Kratzke, N. The #BTW17 Twitter Dataset - Recorded Tweets of the Federal Election Campaigns of 2017 for the 19th German Bundestag. *Data* **2017**, *2*. https://doi.org/10.3390/data2040034.
17. Kratzke, N. Monthly Samples of German Tweets, 2022. https://doi.org/10.5281/zenodo.2783954.
18. Wiggins, A. The Twelve-Factor App, 2017. https://12factor.net.
19. The Kubernetes Authors. Kubernetes, 2014. https://kubernetes.io.
20. Sigelman, B.H.; Barroso, L.A.; Burrows, M.; Stephenson, P.; Plakal, M.; Beaver, D.; Jaspan, S.; Shanbhag, C. Dapper, a Large-Scale Distributed Systems Tracing Infrastructure. Technical report, Google, Inc., 2010.
21. Usman, M.; Ferlin, S.; Brunstrom, A.; Taheri, J. A Survey on Observability of Distributed Edge & Container-based Microservices. *IEEE Access* **2022**, pp. 1–1. https://doi.org/10.1109/ACCESS.2022.3193102.
22. Chow, M.; Meisner, D.; Flinn, J.; Peek, D.; Wenisch, T.F. The Mystery Machine: End-to-end Performance Analysis of Large-scale Internet Services. In Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14); USENIX Association: Broomfield, CO, 2014; pp. 217–231.
23. Bento, A.; Correia, J.; Filipe, R.; Araujo, F.; Cardoso, J. Automated Analysis of Distributed Tracing: Challenges and Research Directions. *Journal of Grid Computing* **2021**, *19*, 9. https://doi.org/10.1007/s10723-021-09551-5.
24. Linux Foundation. Cloud-native Computing Foundation, 2015. https://cncf.io.
25. The OpenTelemetry Authors. The OpenTelemetry Specification - Logs Data Model, 2021. https://opentelemetry.io/docs/reference/specification/logs/data-model/.
26. The OpenTelemetry Authors. The OpenTelemetry Specification - Metrics SDK, 2021. https://opentelemetry.io/docs/reference/specification/metrics/sdk/.
27. The OpenTelemetry Authors. The OpenTelemetry Specification - Tracing SDK, 2021. https://opentelemetry.io/docs/reference/specification/trace/sdk/.
28. The OpenTelemetry Authors. The OpenTelemetry Operator, 2021. https://github.com/open-telemetry/opentelemetry-operator.
29. Service Mesh Interface Authors. SMI: A standard interface for service meshes on Kubernetes, 2022. https://smi-spec.io.
30. Kratzke, N. log12 - a single and self-contained structured logging library, 2022. https://github.com/nkratzke/log12.
31. Istio Authors. The Istio service mesh, 2017. https://istio.io/.