Journal of Edge Computing, **2024**, 3(1), pp.87-109 https://doi.org/10.55056/jec.728

Telemetry to solve dynamic analysis of a distributed system

Oleh V. Talaver[1], *Tetiana A.* Vakaliuk[1,2,3,4]

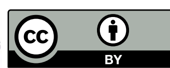
[1]*Zhytomyr Polytechnic State University, 103 Chudnivsyka Str., Zhytomyr, 10005, Ukraine*
[2]*Institute for Digitalisation of Education of the NAES of Ukraine, 9 M. Berlynskoho Str., Kyiv, 04060, Ukraine*
[3]*Kryvyi Rih State Pedagogical University, 54 Universytetskyi Ave., Kryvyi Rih, 50086, Ukraine*
[4]*Academy of Cognitive and Natural Sciences, 54 Universytetskyi Ave., Kryvyi Rih, 50086, Ukraine*

Abstract. In the modern software development world, implementing distributed solutions has become quite common due to the flexibility it brings to big companies. The downside is that when developing such systems, especially in many teams, global design problems may not be obvious and lead to a slowdown in the development process or even problems with the location of errors or degradation of overall system performance. In addition, the timely reaction to system degradation is complicated by the distributed nature of the architecture; while manually configuring rules for reporting problematic situations can be time-consuming and still incomplete, automatic detection of possible system anomalies will give engineers (especially Software Reliability Engineers) the focus on problems. For this reason, applications that can dynamically analyse the system for problems have great potential.
Currently, the topic of using telemetry for system analysis is actively studied and gaining traction, so further research is valuable. The work aims to theoretically and practically prove the possibility of using telemetry to analyse a distributed information system and detect harmful architectural practices and anomalous events. To do this, firstly, a detailed overview of the problems related to the topic and the feasibility of using telemetry is provided; the next section briefly describes the history of the development of monitoring systems and the key points of the latest OpenTelemetry standard, reviews popular application performance monitoring systems, and defines innovative features to be further researched. The main part includes an explanation of the approach used to collect and process telemetry, a reasoning behind the usage of Neo4j as a data storage solution, a practical overview of graph theory algorithms that help in the analysis of the collected data, and a description outlining how the PCA algorithm is employed to detect unusual situations in the whole system instead of individual metrics. The results provide an example of using the software presented with Neo4j Bloom to visualise and analyse the data collected over several hours from the OpenTelemetry Demo test system. The last section contains additional remarks on the results of the study.[1]

Keywords: distributed systems, microservices, dynamic analysis, architectural smells, anti-pattern, visualization, telemetry, anomalies, Open Telemetry, graph theory, statistical analysis

[1]This article is an extended version of a conference talk presented at the 6th Workshop for Young Scientists in Computer Science & Software Engineering (CS&SE@SW 2023) [29].

olegtalaver@gmail.com (O. V. Talaver); tetianavakaliuk@gmail.com (T. A. Vakaliuk)
http://acnsci.org/vakaliuk/ (T. A. Vakaliuk)
0000-0002-6752-2175 (O. V. Talaver); 0000-0001-6825-4697 (T. A. Vakaliuk)



© Copyright for this paper by its authors, published by Academy of Cognitive and Natural Sciences (ACNS). This is an Open Access article distributed under the terms of the Creative Commons License Attribution 4.0 International (CC BY 4.0), which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

1. Introduction

In recent years, distributed architectures such as microservices have received much attention and popularity due to the opportunities that the architectural pattern opens up in terms of optimization, technology stack diversification, and more [5]. When built correctly, distributed systems simplify the development process when many teams are involved, reduce the complexity of changes or the dependence of teams on each other, and speed up development. Other commonly mentioned advantages such as the reliability of a whole system when separate components go down and easiness of understanding (dubious due to the scattered nature of the use case logic) – are secondary because of the additional complexity that microservices bring [27] slows down the development of such a system. In contrast, the general advantage of technological heterogeneity only increases the effort required to maintain the codebase [3, 28, 32]. On the other hand, while prone to losing the overall application structure and distinct module separation, monolith architecture is preferable at the start of new project development when there is much uncertainty involved and frequent global changes are required. Physical boundaries between services complicate the refactoring process because of the distributed nature of the application state and dependencies. Therefore, more tools are needed to analyze the system and respond to problems. The development of distributed information systems requires more effort, especially when it comes to monitoring the entire system and finding problematic areas [27], because, unlike a monolith, such a system has many components developed in parallel, which may have structural flaws [13, 25], also referred to as architectural smells – design decisions that hinder maintainability and extensibility. For global problems location, an information system analysis is often conducted to find and quickly address such shortcomings [21]. A couple of approaches exist, such as static analysis of the codebase of each system component or analysis of the system logs. Both options are complex because they require adjustment for each system, technology, and programming language. However, the static approach, unlike the dynamic one, allows you to analyze the system without the need to run the whole system, which allows you to correct some local code smells but not the problems of the system as a whole due to the low accuracy and insufficient information about the runtime behaviour [4]. At the same time, dynamic analysis is based on the information gathered in runtime, presenting a more accurate representation of the system utilization. The most prominent dynamic analysis approach is telemetry, which combines three pillars of system observability: logs, metrics, and traces. Therefore, the purpose of the work is the theoretical and practical substantiation of the possibility of using the OpenTelemetry standard to analyze a distributed information system: detecting and quickly responding to harmful architectural practices and anomalous events. Next, we outline the tasks:

- research of state-of-art approaches of telemetry analysis;
- modeling extract, transform and load (ETL) and further telemetry analysis process;
- analysis of the received data to identify harmful practices and anomalies.

2. Theoretical background

The topic of system observability is far from new. In the world of distributed systems, Google is considered a pioneer in the study of the topic of observability. In 2010, Google engineers published a paper called "Dapper – a Large-Scale Distributed Systems Tracing Infrastructure" [24], which prompted the emergence of the first systems for request trace visualization: Jaeger and Zipkin. However, these applications solved the same problem while being incompatible, causing vendor lock, so over time, the development of the OpenTracing [30] standard began. The new standard provides a layer between the application and monitoring systems to track and collect requests. This standard did not solve the whole problem, so the OpenCensus standard was later developed to focus on system metrics and logs collection but also included an alternative implementation of trace collection, which ultimately created more problems, as developers now had to choose between the two standards. For this reason, in 2019, both standards were combined into OpenTelemetry [15] to solve the following tasks:

- gathering traces, metrics, and logs in one place;
- finding anomalies through charts;
- finding the location and cause of anomalies through the review of problematic request traces.

Also, at that time, quite a lot of applications helping in system monitoring had already been presented on the market; for this reason, one of the tasks was to maintain compatibility with them, so the latest standard provides a specification that describes approaches for metrics collection, conventional descriptions of processes such as interaction using the HTTP protocol, RPC [19] without forcing vendor lock. As a result, OpenTelemetry is currently the most active project of the Cloud Native Computing Foundation [18].

One of the central notions introduced in the standards is telemetry – a set of metrics, logs and, most importantly, traces [14], which, in the case of our topic, can be used to build a system model in the form of a directed graph [4] and later used to analyze and identify bad practices and problem areas of the system. The OpenTelemetry standard is relatively new, so there is still active research on the possible use cases, but the central area of use is the visualization of requests (figure 1) with the ability to search for problematic areas, for example, the cause of poor service performance or the root cause of an incorrectly working business process [14].

The idea of using telemetry to improve the structure of a system can be traced back to several research papers released in recent years [6, 7, 20], and has a relatively small list of problems that can be identified, which opens up opportunities for further study of this topic [21].

To understand the scope of future research, a review of popular telemetry visualization and application monitoring solutions was conducted. The critical overview is presented below.

Signoz is a relatively new open-source product offering mostly basic monitoring capabilities but with more details than other open-source solutions. It supports advanced filtering and customizable dashboards, enables notifications and has simple system graph visualization, where service dependencies, error and request rates are shown in the figure 2.

ServiceNow Cloud Observability is a closed platform offering comprehensive multi-functional metrics analysis and charting capabilities. It includes a correlation engine that allows detecting

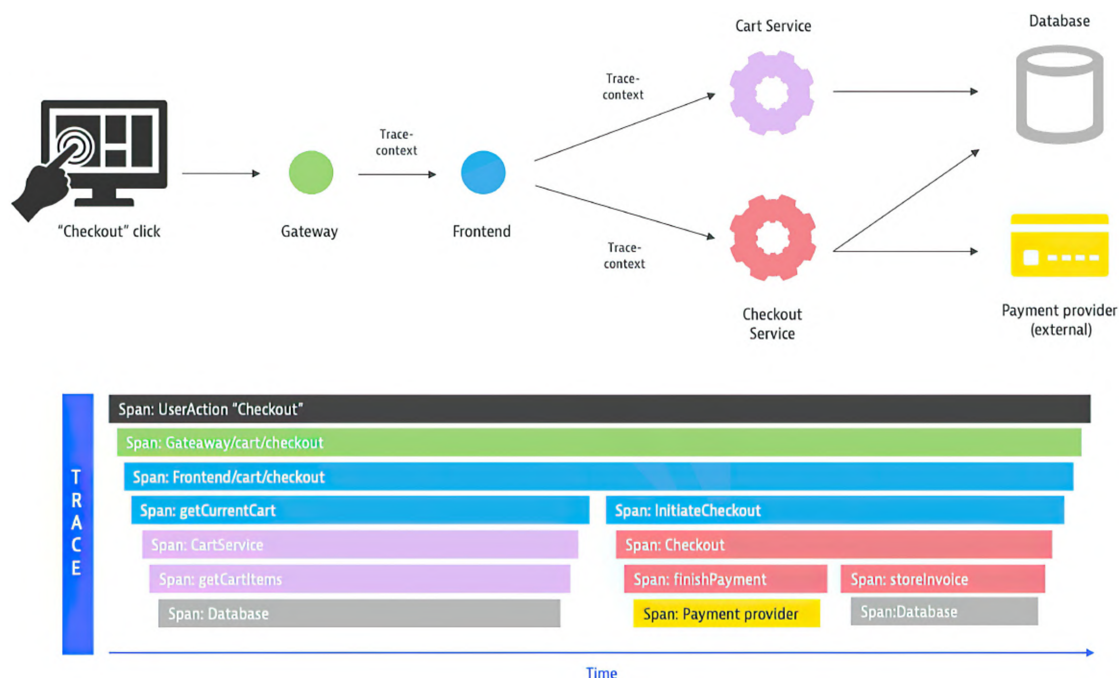


Figure 1: Request trace visualization.

and analyzing anomalies by comparing a problematic period of time with a base one and finding differences in attributes among the equivalent trace spans.

Honeycomb (figure 1) focuses on identifying the causes of abnormal situations in the system, which helps in finding performance problems much quicker. Otherwise, it has standard capabilities like system map visualization with some filters, alerting, and notification.

New Relic is an enterprise-level application monitoring system with numerous capabilities, from monitoring to anomaly analysis. Compared to previous systems, anomaly analysis is automatic and is included in many places to show differences between groups of services, similar requests, and degradation of performance and quality. The analysis considers load seasonality to exclude expected spikes from results.

Based on the review above, the features found were split into two categories: base features, industry standard for such systems and innovative features – valuable capabilities that make systems stand out.

A comparison of the applications' capabilities is shown in table 1.

Base features include:

- list of system services with key metrics shown for each separately (percentage of errors for some time, percentiles of service request execution speed);
- the possibility of building dashboards with custom queries;
- review of problems and exceptions encountered in system requests. In more advanced systems, issues management and collaboration capabilities are present;

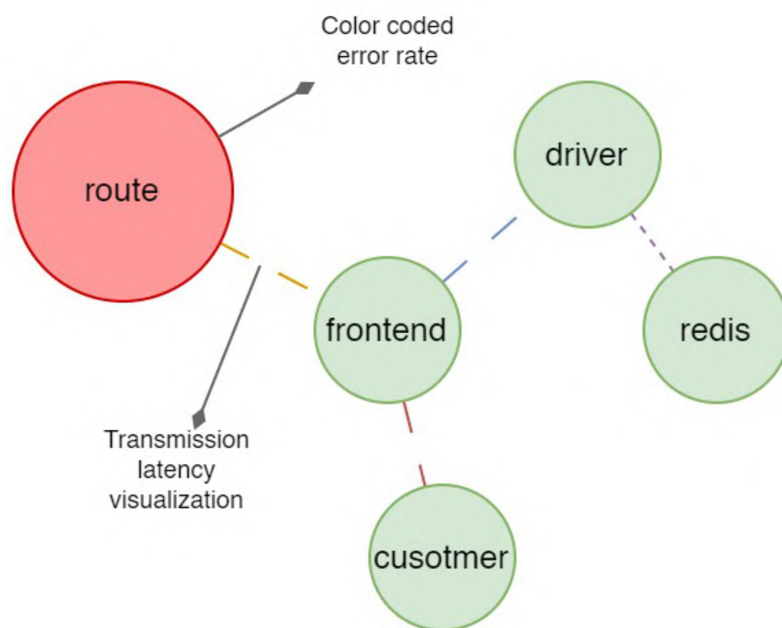


Figure 2: Signoz system graph (visual reconstruction of the result to improve readability).

Table 1
Application features comparison.

Application	Jaeger	Signoz	Aspecto	ServiceNow	Honeycomb	New Relic
Services, metrics visualization	−	+	−	++	+	++
Exceptions management	−	+	−	−	−	++
Search and visualization of traces	+	+	++	++	++	++
System graph visualization	−	+	−	+	++	++
Alerts management	−	+	−	+	++	++
Automatic anomaly detection	−	−	−	−	−	++
Third-party systems integrations	−	−	−	++	+	++
Root cause analysis	−	−	−	+	+	++

- filtering and viewing traces visualization. Such views usually include a couple of representations like a request graph that shows the path and services involved, various diagrams to show the time that was spent in a particular service;
- alerts management. Usually, the use case implies calculation of the compliance rate for

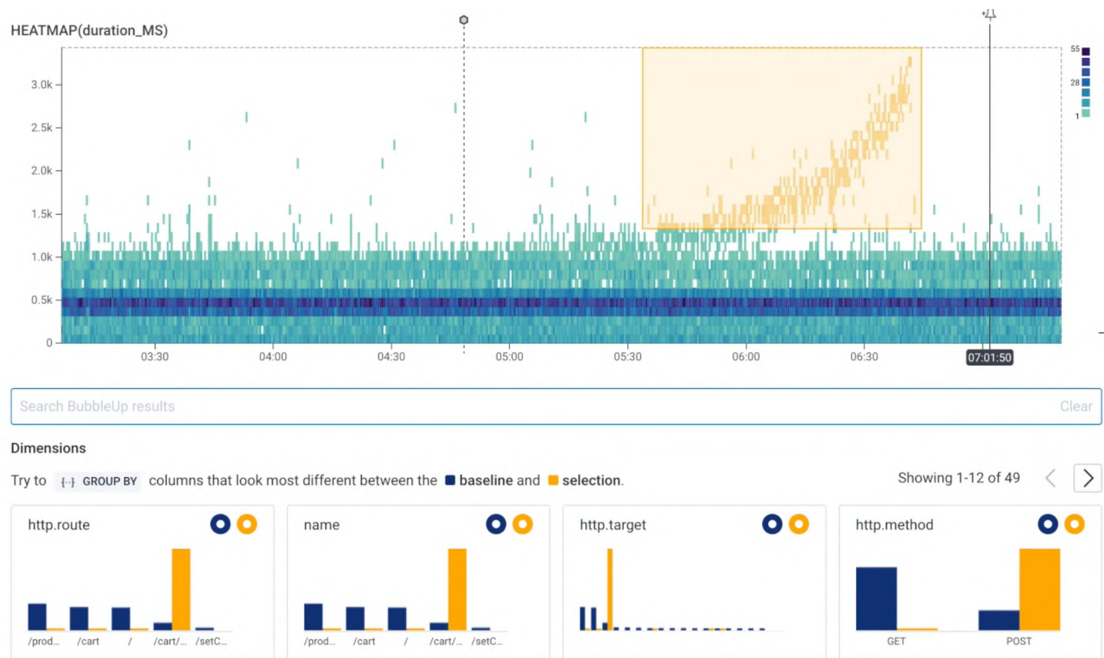


Figure 3: Honeycomb bubble up – correlation engine.

SLO measurements, notification;

- system graph visualization may be comprehensive and include multiple layers for visualization of physical relations (part of a particular node, pod…) and logical with call dependencies.

Innovative features include:

- comparison of groups of request traces to find factors contributing to changes in performance. The example may be that very slow queries are seen for some small portion of users that use additional parameters;
- automatic detection of problems in the system (anomalies detection). More advanced implementations may include the ability to adapt to expected changes, for example, day and night load difference, increased usage during some period of a day;
- root cause analysis – automatic determination of the causes of problems in the system pointing to a problematic service, endpoint, or release;
- integration with the infrastructure – to display more information about the location of service instances more server related metrics;
- integration with external systems such as GitHub, Continuous Integration (CI), and Continuous Delivery (CD) platforms to be able to quickly jump from one system to a contextually related place of another (e.g. file source), collect deployment events, track service versions, display additional metadata related to the services that are stored centrally in the repository.

After analyzing the above functionality, two areas of improvement were found:

- identification of bad architectural practices that cause a big problem in distributed systems development because when individual teams work on separate parts of the application, it may be tough to track dependencies and see the bigger picture, which leads to degradation of performance and maintainability. The architectural smells that will be visualized are the following:
  - bottleneck – a component on which many other components depend using synchronous requests. This may lead to system fragility when this component is unavailable;
  - cyclic dependency – a cluster of components highly dependent on each other, causing high coupling. This practice indicates incorrectly separated responsibilities of the components;
  - nano-service – a service that depends on many others through synchronous requests. Often, this means the service is too small but still requires efforts to support, not to mention the overhead the synchronous requests add due to slower network speed compared to in-process invocation;
- anomalies detection in the whole system – analysis of key metrics of system components to find problematic areas. Compared to other available solutions, the current implementation will consider the whole system instead of separate requests and use cases.

3. Methods

3.1. Defining data and storage for architectural smells detection

The proposed analytical system receives a constant stream of telemetry data and aggregates it by updating the system model in the form of a directed graph stored in a graph database management system (DBMS). Then, the model can be used for analysis, searching for structural anti-patterns.

Constructing the system's graph model involves processing traces of requests (figure 4). Once they are received, the process creates or updates information about available resources (services, storages, proxies) and stores information about changes in the storage. Operations available in the service (operation) and individual sub-requests (hop).

To build a system graph for further analysis, the data storage must have the following information:

- *resources* are interacting components of the system. A resource must have a name, type (service, storage), date of creation and last use;
- *operations* are defined by one resource and called by other ones; have statistics on the number of calls, errors, the last date of creation, and use;
- *calls* – connections between a resource and an operation. They have the creation date, last use, type (synchronous, asynchronous), and number of errors.

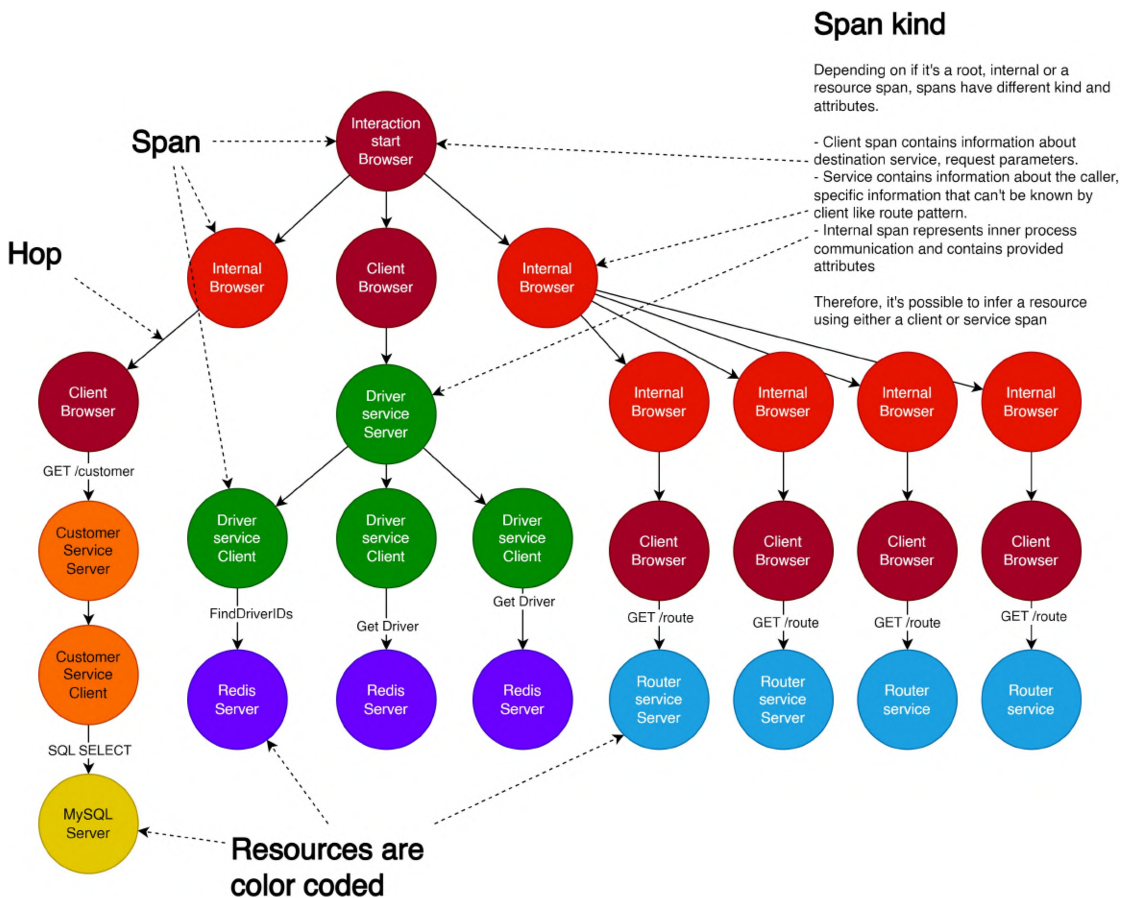


Figure 4: Example of a request tree.

Neo4j was chosen as the storage of the system model since it physically stores data as a graph, which makes it possible to use graph traversal algorithms to find bad practices in the system, namely:

- clustering coefficient – measures the degree of vertex connectivity; will help show service groups in the system [11];
- degree centrality – measure the number of connections between vertices; makes it possible to calculate the affinity (coupling) metrics of components in the system [10];
- strongly connected components – finds groups where each vertex is accessible from any other; helps to identify cyclic dependencies in the system [12].

The graph DBMS structure is presented in figure 5.

Data storage has two types of nodes: resources and operations. Resources are related to the operations with the "Provides" relation. To show calls, the "Calls" relationship is used, which aggregates statistics for all identical calls from one resource to an operation of another resource (figure 6).

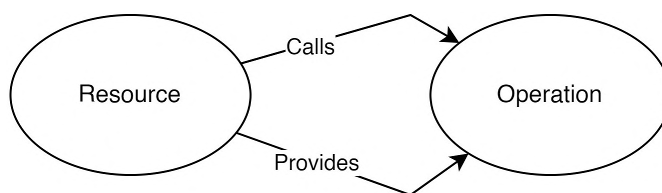


Figure 5: Simplified diagram of the structure of a graph DBMS.

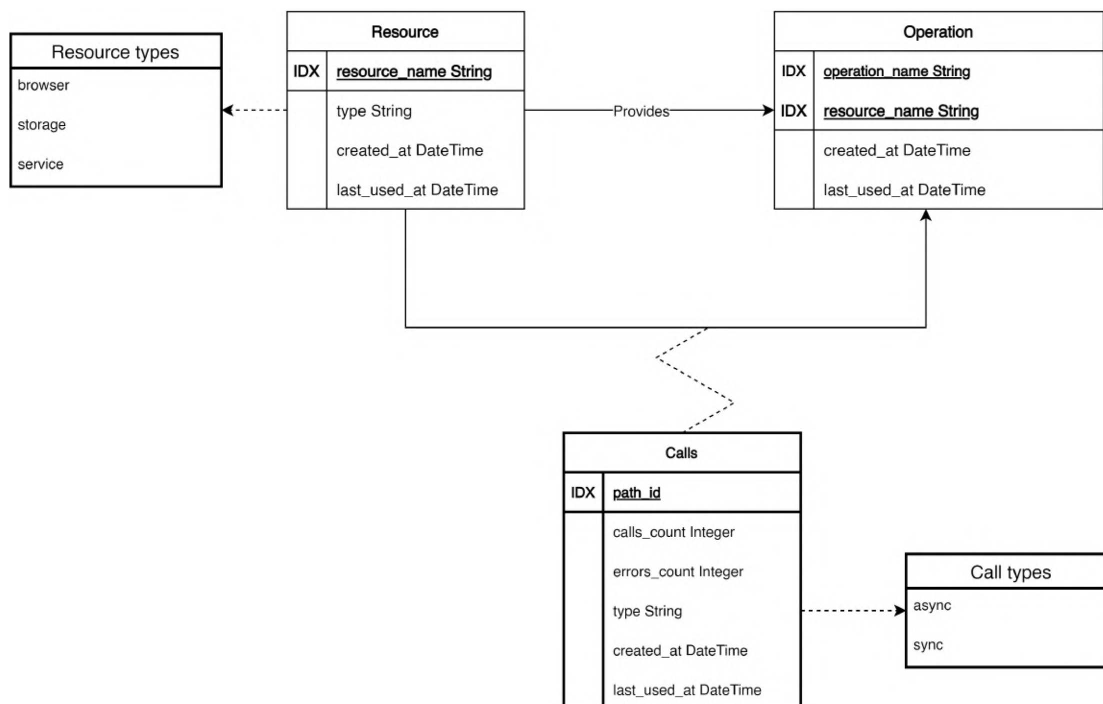


Figure 6: Complete graph DBMS structure diagram.

The ETL process begins with the system's instrumentation – the installation of modules for popular libraries that will collect the telemetry and manual changes in service code to provide more details of a particular process in the system. Later, the telemetry is sent to the OpenTelemetry Collector [16] – a separate modular application developed by the authors of the standard, which allows you to unify the process of collecting, transforming, and exporting telemetry into various popular monitoring systems. In figure 7, we can see how metrics (blue) and traces (red) are emitted and occasionally sent by every service in the open telemetry demo project to the collector. Telemetry is received by modules called "receivers", which can receive or extract data from various systems, like Jaeger and Prometheus. However, the OTLP protocol is developed explicitly for telemetry transportation in this case. In the collector, there are two other modules: processors, which help to transform and filter telemetry and exporters, which send the telemetry to external systems. There are numerous available modules, but our task is

to create a custom exporter that takes batches of traces, extracts necessary data, and unloads it into neo4j.

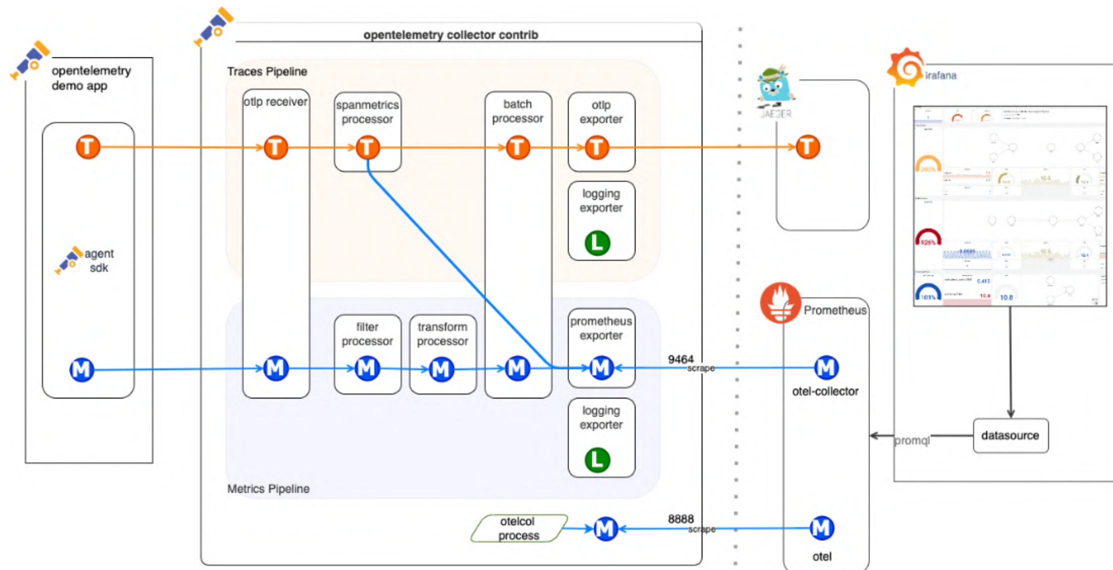


Figure 7: Telemetry flow diagram.

The developed module takes a group of trace objects as input (the detailed structure of a trace with an explanation can be found in the standard source code [31]) and loops through each span. A span defines some operation in the system (see figure 4); it can be the start of the service operation ("server" span for direct request, "consumer" for async events handling), call another server operation ("client" span for a request, "producer" for async event) or in process operation (internal span). Since the trace is a chain of consecutive spans, all but the first root spans have a parent. Following the chain, we can distinguish individual operations, resources, and calls. All this data is inserted into the database as follows (snippet of a Cypher request for upserting a resource in the system):

```
...
OPTIONAL MATCH (r:Resource)
  WHERE $toResourceName =~ "(?i).*" + r.name + ".*"
  OR r.name =~ "(?i).*" + $toResourceName + ".*"
WITH CASE r IS NULL
       WHEN true THEN $toResourceName
       ELSE r.name
       END as resourceName, operation
MERGE (toResource:Resource {name: resourceName})
  ON CREATE
  SET
  toResource.createdAt =  timestamp(),
```

```
  toResource.lastUsedAt = timestamp(),
  toResource.type = $toResourceType
  ON MATCH
  SET toResource.lastUsedAt = timestamp()

WITH operation, toResource
...
```

3.2. Methods of anomalies detection

The problem of finding and analyzing anomalies is quite common in computer science and often varies depending on the domain in which the analysis takes place. For example, when reading data from sensors for further analysis, it is essential to find and correct outliers. When analyzing a business process, it is sometimes necessary to find unusual events to analyze what led to them. In software reliability engineering, the topic of mean time to detect is one of the most critical indicators because if the problem is found earlier, it is fixed earlier.

Analysis of anomalous changes is already present, at least in New Relic. However, it is present at the level of individual services, not the entire system. Although there is not enough data to confirm this, the platform analyzes metrics, including key metrics, using the Exponential Smoothing [1], which is a method of predicting a single variable and, depending on the type, can take into account seasonality [2]. However, it is also possible to find anomalies opposite – from a larger scale, using multivariate algorithms, which will be used in this work.

An anomaly is an abnormal situation defined as a substantial difference between expected and actual measurements. Therefore, the process of finding an anomaly includes the process of predicting the value of a particular measurement based on historical data [8].

The problem of finding anomalies in multivariate datasets is quite popular and critical because little to no measurements are univariate [26].

Algorithms are divided into the following training approaches:

- unsupervised – the dataset used for model training does not include labels indicating anomalous situations;
- semi-supervised – the dataset has anomalous situations labelled;
- supervised – the whole dataset is labelled, the least commonly used type of algorithm, as it is difficult to get fully labelled data.

Due to the difficulty of obtaining labelled data, unsupervised models are the most popular. At the same time, it is also possible to add the possibility of providing feedback and a correction loop when using models for semi-labelled datasets. As part of this work, the unsupervised model is reviewed. While "None of the unsupervised methods is statistically better than the others" [8], which is due to the complexity of training on unlabeled data in which extra parameters only interfere, it was decided to choose Principal Component Analysis (PCA) – a statistical method of multivariate analysis used to identify the main structural components in a dataset. The main goal of PCA is to reduce the dimensionality of data while explaining the dataset in as much detail as possible, which, due to the simplicity of the approach, is well-suited for multivariate datasets and makes it the most common algorithm.

Essentially, PCA converts the initial correlated variables into new linear combinations called principal components. The first principal component is defined in such a way that it explains the most significant part of the data variance. Each successive principal component is chosen to be orthogonal to the previous ones and explain the residual variance as much as possible.

In the case of identifying anomalies in the system, we are interested in the following information:

- calls – number of incoming, outgoing, and internal calls (synchronous and asynchronous when using a queue or other message brokers) with and without errors;
- duration – time spent processing requests.

To obtain the necessary data in metrics and group collected data, you need to use a unique connector component that transforms traces into call and duration metrics. Thus, the collector receives information about the request via the Open Telemetry Protocol (OTLP) and then groups and extracts the necessary metrics to export later.

Each of the system's components (resources) collects metrics for a certain period. Metrics have different types of values. For example, the number of calls has the sum type, which is a counter of certain events for a period and, in this case, is a monotonous sequence because the number of calls never decreases.

It is also important to note that the metrics are returned as a delta (the value of aggregationTemporality is 1) and not a cumulative value because we are interested in the number of calls in a certain period, not the absolute value. Each metric can have multiple points that represent different attribute-defined dimensions (dimensions are customizable), so separate counters have been set up for different request types (span.kind) and statuses (status.code).

Calls duration metric snippet:


```
{
"name": "duration",
"unit": "ms",
"histogram": {
"dataPoints": [
{
"attributes": [
{
"key": "service.name",
"value": {
"stringValue": "quoteservice"
}
},
...
],
"startTimeUnixNano": "1685509043760171242",
"timeUnixNano": "1685509058790174364",
"count": "1",
```

```
"sum": 0.006665,
"bucketCounts": [
"1",
"0",
...
],
"explicitBounds": [
0.1,
1,
...
],
"exemplars": [
{
"timeUnixNano": "1685509058790174364",
"asDouble": 0.006665,
"spanId": "ade03fcb73f18048",
"traceId": "7f6cf387237813d1f3891b5f21b09be2"
}
]
},
],
"aggregationTemporality": 1
}
}
```


If we take the call duration metric, then in this case we have a histogram, which is a certain aggregation of values and their distribution over intervals used for easier visualization.

But in this form, we will not be able to use this data. Firstly, all the metrics for individual services are separated (figure 8) and converted to time series (figure 9) to later be combined based on timestamp (figure 10).

results
accountingservice_incomingAsync_request_duration.csv
accountingservice_incomingAsync_requests_without_errors.csv
adservice_internal_request_duration.csv
adservice_internal_requests_without_errors.csv
adservice_outgoing_request_duration.csv
adservice_outgoing_requests_without_errors.csv
checkoutservice_incoming_request_duration.csv
checkoutservice_incoming_requests_without_errors.csv
checkoutservice_internal_request_duration.csv

Figure 8: Results of a metric timeseries per service extraction .

```
time,value
1685509058790506925,3.201241
1685509073760723132,86.654854
1685509088760464981,4.221619
1685509103760203727,4.63424825
1685509118760201716,4.838916
1685509133766908850,45.745556
1685509163765380527,3.4242303333333335
1685509178765006382,4.1387815
1685509193764866922,10.35692
1685509208765251617,2.4504515
1685509223764207083,3.4219543333333333
1685509238764558486,15.992556
1685509253763748445,3.518201
1685509268764675932,6.9955475
1685509283764021324,3.0531842499999997
1685509298763192098,2.2964245
1685509313763421062,3.841479666666667
```

Figure 9: Metric timeseries file example.

11 rows 161 rows × 54 columns pd.Dataframe CSV

time	frauddetectionservice_incomingAsync_request	currencyservice_outgoing_requests_without_	adserv
2023-05-31 04:57:00	0.141367	4.500000	
2023-05-31 04:58:00	0.227574	6.500000	
2023-05-31 04:59:00	0.191880	6.333333	
2023-05-31 05:00:00	0.287567	5.000000	
2023-05-31 05:01:00	0.135161	9.500000	
...	...	...	
2023-05-31 07:33:00	0.094923	9.000000	
2023-05-31 07:34:00	0.076495	7.333333	

Figure 10: Combined metrics.

From the intermediate results, you can clearly see the correlation between the different metrics of the system components (figure 11), which is confirmed by a correlation map (figure 12).

The process of identifying anomalies occurs by splitting the data sample into two periods, the first is used to train the PCA statistical model, the second is used to compare with the predicted values obtained from the model and, estimate the error for all and specific metrics.

Snippet of model training:

```
from sklearn.model_selection import TimeSeriesSplit
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
import numpy as np

rec_errors_samples = {}
rec_errors_features = {}
```

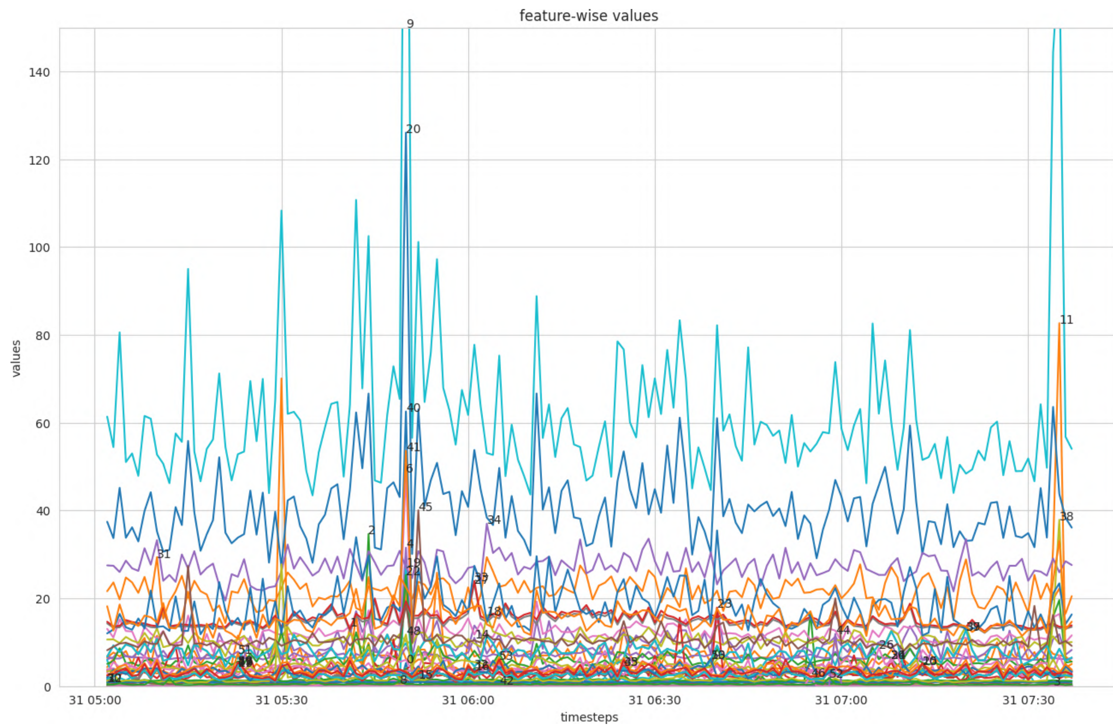


Figure 11: Chart of metric values over time.

```
for i, (past_id,future_id) in enumerate(
        TimeSeriesSplit(10).split(df)
):
    scaler = StandardScaler()
    pca = PCA(0.7, random_state=33)
    pca.fit(scaler.fit_transform(df.iloc[past_id]))

    df_inverse = pca.inverse_transform(
        pca.transform(
            scaler.transform(df.iloc[future_id])
        )
    )
    time = df.iloc[past_id[-1]].name
    diff = scaler.transform(df.iloc[future_id]) - df_inverse
    rec_errors_samples[time] = np.linalg.norm(diff, axis=1)
    rec_errors_features[time] = np.linalg.norm(diff, axis=0)
```

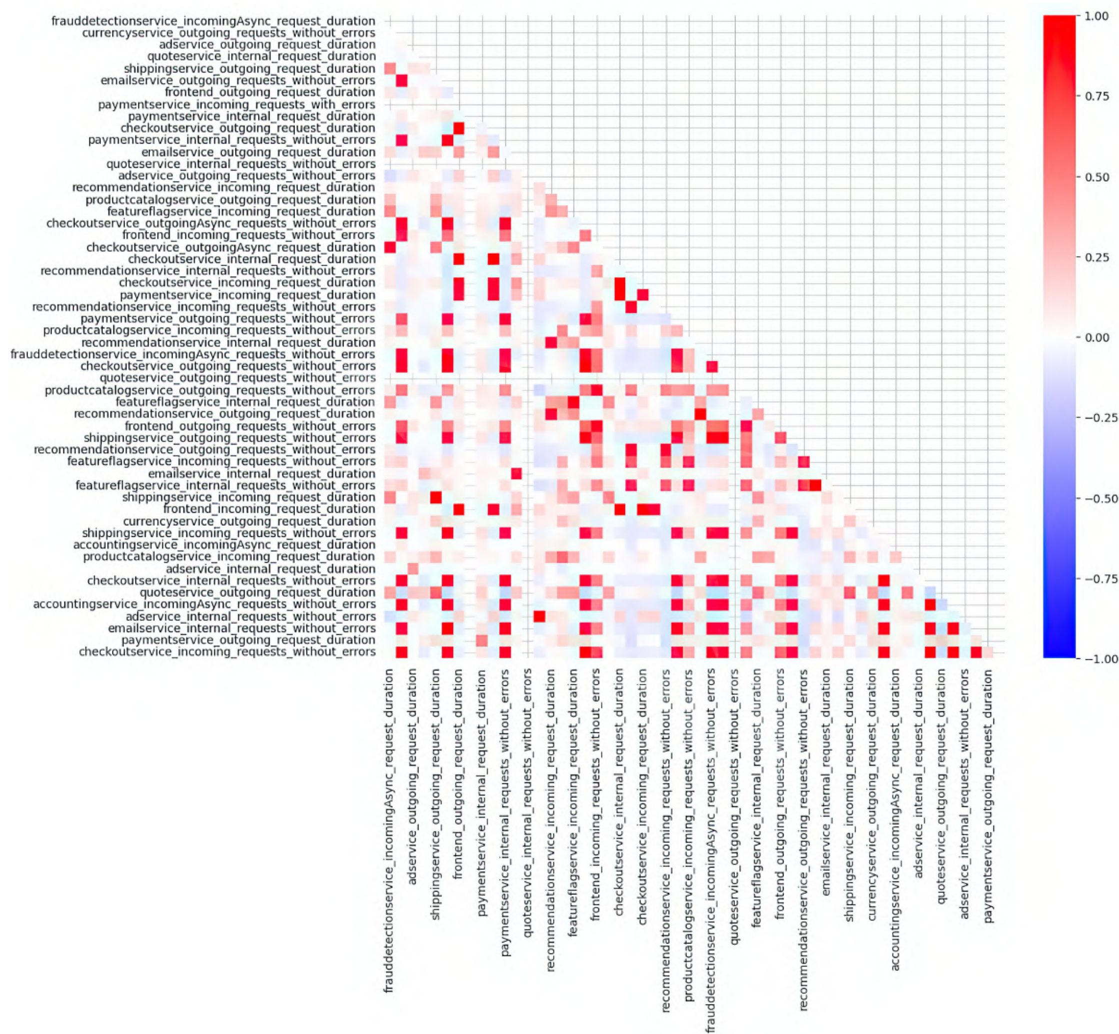


Figure 12: Metrics correlation map.

4. Results

The OpenTelemetry Demo project was used as a test system [17], specially designed for testing applications working with telemetry. This distributed system has components built with different technologies and is automatically loaded using a load generator service.

4.1. Visualization of the service graph using Neo4j tools

After running the whole system, the graph database has the following data (figure 13). You can see that the graph has many nodes with the type of operation (orange circles) and slightly fewer services (purple circles). You can see the “calls” and “provides” relationships depicted as arrows between them.

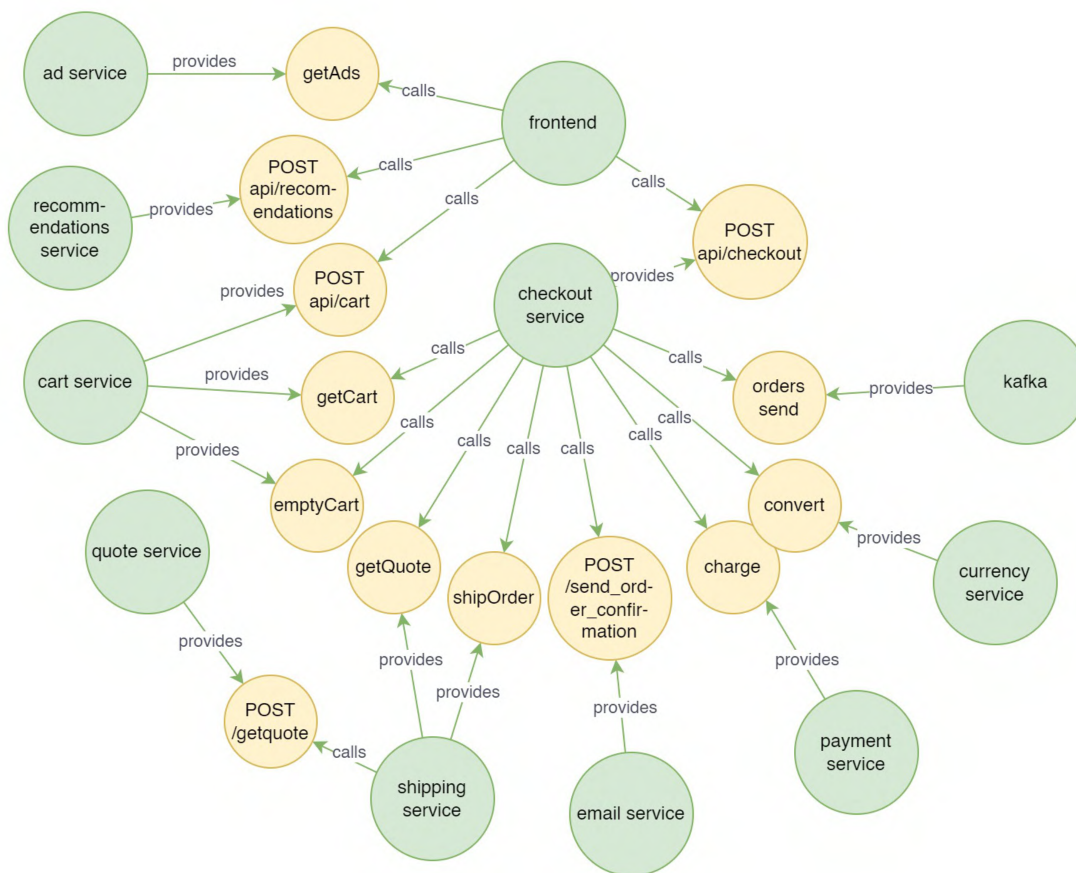


Figure 13: Visualization of the full graph of services, operations and connections between them using Neo4j Browser (visual reconstruction of the result to improve readability).

To simplify the graph, a function from the APOC library is used [9] for Neo4j in order to visualize the graph projection and show service dependencies (figure 14).

A snippet of a virtual relationship visualization query:

```
MATCH (r1:Resource)-[:Calls]->(:Operation)<-[:Provides]-(r2:Resource)
RETURN r1, r2, apoc.create.vRelationship(r1,'DependsOn',{}, r2) as rel
```

In figure 14, you can see the dependence of the checkout service on many others. To confirm this, let us use Neo4j Bloom to visualize Local Clustering Coefficient [11] and Degree Centrality [10] algorithms.

In the resulting diagram (figure 15), clusters are marked with distinct colours, and their size indicates the dependence of services on peers. From the diagram, it is also clear that the checkout service has many dependencies. This way, you can quickly analyze the application's architecture and see parts that must be refactored to prevent the whole application halts due to a single bottleneck component.

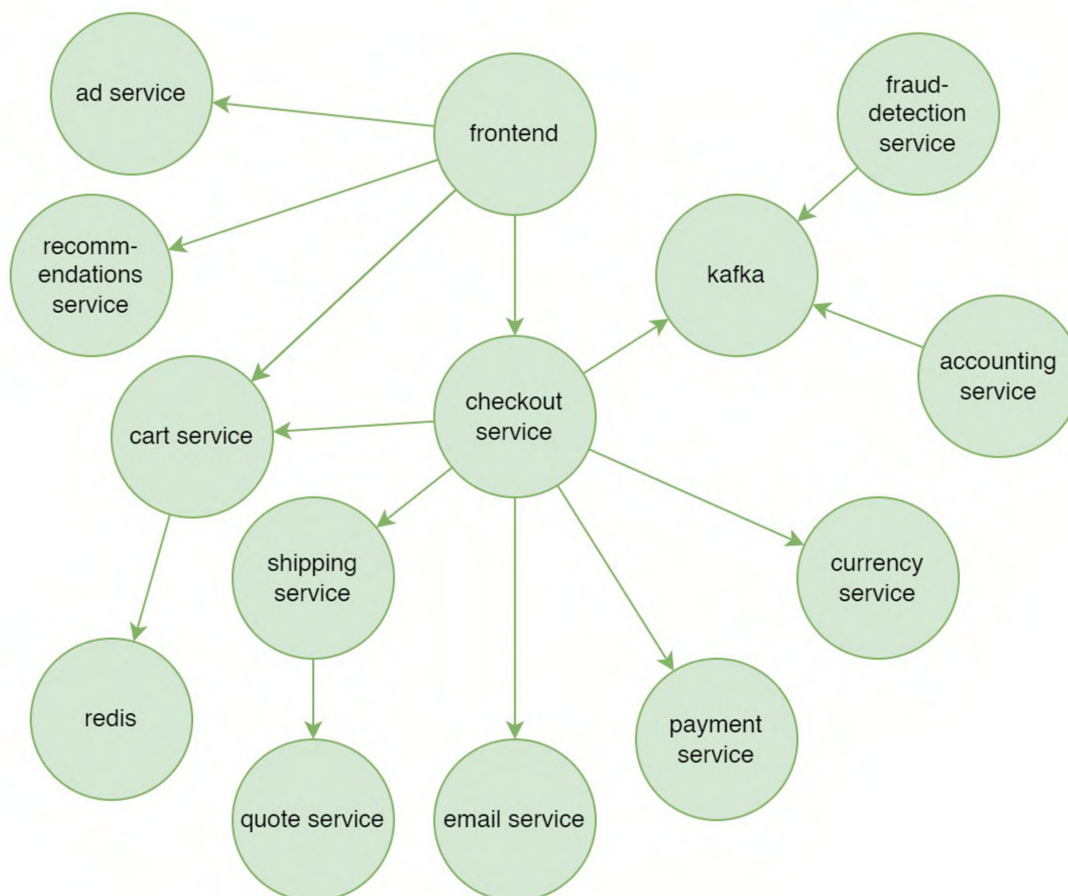


Figure 14: Visualization of the dependency graph in Neo4j Browser (visual reconstruction of the result to improve readability).

4.2. Time interval anomalies analysis

A few hours-long time interval was chosen to detect anomalies. It has been processed using the PCA algorithm, and after receiving errors for each time point, a visual analysis can be performed for the presence of spikes in the error values (figure 16).

As you can see in the plot, between 6:50 a.m. and 7 a.m., there were some changes that led to a relatively big error. From the error graph for each of the features, it can be seen that feature 44 is involved in this error, so by conducting a more detailed analysis of the values of this metric, we can see that all values are kept near 0, while there is an outlier with a value of about 12.

5. Discussion

Compared to static analysis approaches, dynamic analysis allows you to see the accurate picture of the entire system, all possible query paths that are used, and accurately indicate the

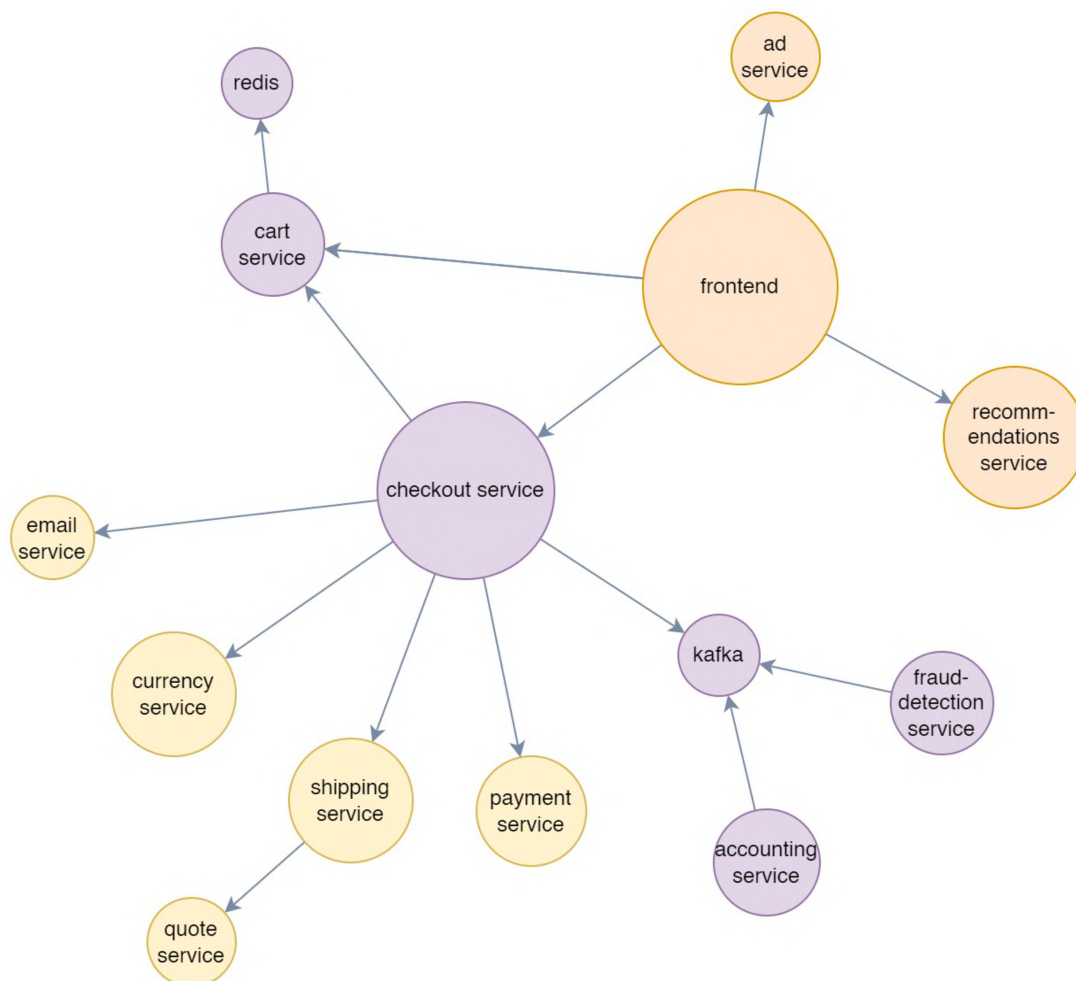


Figure 15: Visualization of the dependency graph of services considering clustering and centrality algorithms in Neo4j Bloom (visual reconstruction of the result to improve readability).

components that cause a problem in the performance of the system at a particular moment, in contrast to static analysis of individual modules, which is better suited for the tasks of identifying code smells. Telemetry, in turn, allows you to combine all key indicators and add the additional context that allows you to get more information for analysis.

The practical use of a simple statistical unsupervised PCA algorithm has demonstrated the possibility of using such a model to identify anomalies, which can significantly simplify the work of engineers because instead of looking at dozens of charts and responding to user messages in support, this statistical analysis suggests the occurrence of anomalous situations in the system automatically. When compared with the approaches of analyzing each metric of the system separately (using appropriate statistical methods, for example, those used in NewRelic [1]), this method gives the general picture, allowing you to understand the situation in the entire system, but also provides the cause of the problem. Compared to supervised algorithms, especially

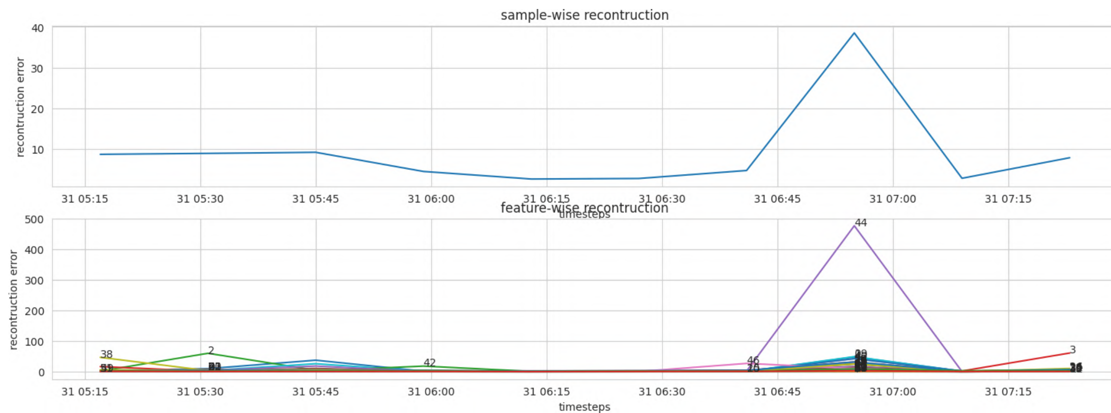


Figure 16: The result of displaying the data reconstruction error for all and individual metrics.

neural networks [22, 23], using the proposed method removes the need to retrain the model to adapt to regular changes (e.g., a natural increase in the number of users of the system), because the analysis takes place in a specific window, although undoubtedly this window should be of a particular size to cover a sufficient amount of data for training and analysis and at the same time not be too sensitive to seasonal changes (for example, activity during the day vs. activity at night), which needs to be tested and determined on a natural system.

6. Conclusions

The paper discusses the use of telemetry for dynamic analysis of the system for anomalous events and architectural smell detection.

An analysis of the problems related to distributed systems development with a detailed summary of the comparison of the monolithic and distributed architectures was carried out, which made it possible to determine the need for applications for monitoring and rapid response to problems in an extensive system. Studies on the use of telemetry for dynamic system analysis, which have been published in recent years, have shown the potential of this approach. The history of the system monitoring topic development and the main aspects of the latest OpenTelemetry standard were reviewed as well as popular applications performance monitoring solutions were compared to later list the features presented in the systems, divide them into groups of essential and innovative, and define the tasks for the study.

Later, the primary data flows required for analysis were identified, and a model of a graph DBMS was built. The model includes the following entities: operations, resources, and relationships, which determine the direction of resource dependence and ownership of operations. After that, telemetry extraction, processing and unloading using the OpenTelemetry Collector was reviewed. The main types of anomaly detection algorithms were studied, and the multivariate PCA statistical method was chosen to analyse unlabeled telemetry data. A custom component of the collector application was developed to transform and insert information into the Neo4j datastore. The necessary features to be used are the process of collecting appropriate numbers and the duration of calls within the system to find anomalies. An algorithm for collecting

metrics was described. The next part overviewed the method of using a statistical model of PCA to identify anomalies.

The next part used an aggregated graph model to analyze architectural smells. Several possible visualizations of the dependency graph using Neo4j and Neo4j Bloom were provided, and clustering and centrality algorithms were used to identify problem areas in the system architecture visually. The statistical model results based on the Principal Component Analysis algorithm were also analyzed. The accuracy of this model is sufficient to determine anomalous events.

The topic of telemetry usage to find bad architectural practices has the potential for further development [7]. After all, the collected data is enough to determine more complex patterns: too long synchronous and asynchronous requests, long chains of synchronous requests, too many different technologies in a small system, and a significant time difference between when an event is published and processed. To provide even more opportunities for analysis, it is possible to enrich system resources with additional metadata indicating belonging to a specific bounded context (to compare de jure and de facto clusters of contexts, to identify situations of using the same database in different parts of the application) [5], dates of any changes or releases. Also, this data should be displayed on the main map of the system components. Further development of anomaly analysis includes integration with an application performance monitoring (APM) system, the ability to configure threshold values for reconstruction error, adding custom metric streams for analysis, and testing on a natural system by comparing it with existing approaches.

Suppose we consider other topics of telemetry usage. In that case, we cannot omit the topic of analyzing individual use cases in the application, which are sets of requests that go through a bunch of services and have variations depending on some stored state of the application; the analysis includes both the ability to evaluate performance, the number of errors of a particular use case, the ability to subscribe a specific development team to updates for a quick response in case of anomalous situations, and ability to view changes, including performance, between different releases.

References

[1] Boone, N.D., 2017. Dynamic Baseline Alerts Now Automatically Find the Best Algorithm for You. Available from: https://newrelic.com/blog/how-to-relic/baseline-alerts-algorithm.

[2] Brownlee, J., 2020. A Gentle Introduction to Exponential Smoothing for Time Series Forecasting in Python. Available from: https://machinelearningmastery.com/exponential-smoothing-for-time-series-forecasting-in-python/.

[3] Bucchiarone, A., Dragoni, N., Dustdar, S., Larsen, S.T. and Mazzara, M., 2018. From Monolithic to Microservices: An Experience Report from the Banking Domain. *IEEE Software*, 35(3), pp.50–55. Available from: https://doi.org/10.1109/MS.2018.2141026.

[4] Cerny, T., Abdelfattah, A.S., Bushong, V., Al Maruf, A. and Taibi, D., 2022. Microservice Architecture Reconstruction and Visualization Techniques: A Review. *2022 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. pp.39–48. Available from: https://doi.org/10.1109/SOSE55356.2022.00011.

[5] Francesco, P.D., Malavolta, I. and Lago, P., 2017. Research on Architecting Microservices:

Trends, Focus, and Potential for Industrial Adoption. *2017 IEEE International Conference on Software Architecture (ICSA)*. pp.21–30. Available from: https://doi.org/10.1109/ICSA.2017.24.

[6] Gamage, I.U.P. and Perera, I., 2021. Using dependency graph and graph theory concepts to identify anti-patterns in a microservices system: A tool-based approach. *2021 Moratuwa Engineering Research Conference (MERCon)*. pp.699–704. Available from: https://doi.org/10.1109/MERCon52712.2021.9525743.

[7] Guo, X., Peng, X., Wang, H., Li, W., Jiang, H., Ding, D., Xie, T. and Su, L., 2020. Graph-Based Trace Analysis for Microservice Architecture Understanding and Problem Diagnosis. *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, ESEC/FSE 2020, p.1387–1397. Available from: https://doi.org/10.1145/3368089.3417066.

[8] Han, S., Hu, X., Huang, H., Jiang, M. and Zhao, Y., 2022. ADBench: Anomaly Detection Benchmark. In: S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho and A. Oh, eds. *Advances in Neural Information Processing Systems*. Curran Associates, Inc., vol. 35, pp.32142–32159. Available from: https://proceedings.neurips.cc/paper_files/paper/2022/file/cf93972b116ca5268827d575f2cc226b-Paper-Datasets_and_Benchmarks.pdf.

[9] Neo4j APOC Library, 2023. Available from: https://neo4j.com/developer/neo4j-apoc/.

[10] Neo4j Degree Centrality, 2023. Available from: https://neo4j.com/docs/graph-data-science/current/algorithms/degree-centrality/.

[11] Neo4j Local Clustering Coefficient, 2023. Available from: https://neo4j.com/docs/graph-data-science/current/algorithms/local-clustering-coefficient/.

[12] Neo4j Strongly Connected Components, 2023. Available from: https://neo4j.com/docs/graph-data-science/current/algorithms/strongly-connected-components/.

[13] Niedermaier, S., Koetter, F., Freymann, A. and Wagner, S., 2019. On Observability and Monitoring of Distributed Systems – An Industry Interview Study. In: S. Yangui, I. Bouassida Rodriguez, K. Drira and Z. Tari, eds. *Service-Oriented Computing*. Cham: Springer International Publishing, pp.36–52. Available from: https://doi.org/10.1007/978-3-030-33702-5_3.

[14] Observability Primer, 2023. Available from: https://opentelemetry.io/docs/concepts/observability-primer/.

[15] OpenTelemetry, 2024. Available from: https://opentelemetry.io/docs/what-is-opentelemetry/.

[16] OpenTelemetry Collector, 2023. Available from: https://opentelemetry.io/docs/collector/.

[17] OpenTelemetry Demo, 2023. Available from: https://github.com/open-telemetry/opentelemetry-demo.

[18] OpenTelemetry Project Journey Report – 2023, 2023. Available from: https://www.cncf.io/reports/opentelemetry-project-journey-report/.

[19] OpenTelemetry Semantic Conventions, 2024. Available from: https://opentelemetry.io/docs/specs/semconv/.

[20] Parker, G., Kim, S., Maruf, A.A., Cerny, T., Frajtak, K., Tisnovsky, P. and Taibi, D., 2023. Visualizing Anti-Patterns in Microservices at Runtime: A Systematic Mapping Study. *IEEE Access*, 11, pp.4434–4442. Available from: https://doi.org/10.1109/ACCESS.2023.3236165.

[21] Pigazzini, I., Fontana, F.A., Lenarduzzi, V. and Taibi, D., 2020. Towards Microservice Smells

Detection. *Proceedings of the 3rd International Conference on Technical Debt*. New York, NY, USA: Association for Computing Machinery, TechDebt '20, p.92–97. Available from: https://doi.org/10.1145/3387906.3388625.

[22] Pilkevych, I.A., Fedorchuk, D.L., Romanchuk, M.P. and Naumchak, O.M., 2023. Approach to the fake news detection using the graph neural networks. *Journal of Edge Computing*, 2(1), p.24–36. Available from: https://doi.org/10.55056/jec.592.

[23] Semerikov, S.O., Vakaliuk, T.A., Mintii, I.S., Hamaniuk, V.A., Soloviev, V.N., Bondarenko, O.V., Nechypurenko, P.P., Shokaliuk, S.V., Moiseienko, N.V. and Ruban, V.R., 2021. Development of the computer vision system based on machine learning for educational purposes. *Educational Dimension*, 5, p.8–60. Available from: https://doi.org/10.31812/educdim.4717.

[24] Sigelman, B.H., Barroso, L.A., Burrows, M., Stephenson, P., Plakal, M., Beaver, D., Jaspan, S. and Shanbhag, C.K., 2010. Dapper, a Large-Scale Distributed Systems Tracing Infrastructure. Available from: https://api.semanticscholar.org/CorpusID:14271421.

[25] Soldani, J., Tamburri, D.A. and Van Den Heuvel, W.J., 2018. The pains and gains of microservices: A Systematic grey literature review. *Journal of Systems and Software*, 146, pp.215–232. Available from: https://doi.org/10.1016/j.jss.2018.09.082.

[26] Suboh, S., Aziz, I., Shaharudin, S., Ismail, S. and Mahdin, H., 2023. A Systematic Review of Anomaly Detection within High Dimensional and Multivariate Data. *JOIV : International Journal on Informatics Visualization*, 7, p.122. Available from: https://doi.org/10.30630/joiv.7.1.1297.

[27] Söylemez, M., Tekinerdogan, B. and Kolukısa Tarhan, A., 2022. Challenges and Solution Directions of Microservice Architectures: A Systematic Literature Review. *Applied sciences*, 12(11), p.5507. Available from: https://doi.org/10.3390/app12115507.

[28] Talaver, O.V. and Vakaliuk, T.A., 2023. Reliable distributed systems: review of modern approaches. *Journal of edge computing*, 2(1), p.84–101. Available from: https://doi.org/10.55056/jec.586.

[29] Talaver, O.V. and Vakaliuk, T.A., 2024. Dynamic system analysis using telemetry. In: S.O. Semerikov and A.M. Striuk, eds. *Proceedings of the 6th Workshop for Young Scientists in Computer Science& Software Engineering (CS&SE@SW 2023), Virtual Event, Kryvyi Rih, Ukraine, February 2, 2024*. CEUR-WS.org, *CEUR Workshop Proceedings*, vol. 3662, pp.193–209. Available from: https://ceur-ws.org/Vol-3662/paper01.pdf.

[30] The OpenTracing Semantic Specification, 2023. Available from: https://opentracing.io/specification/.

[31] Trace source code, 2023. Available from: https://github.com/open-telemetry/opentelemetry-proto/blob/0a743e76ddbb34d7d46a4c3ca8f9d7bdbb81e389/opentelemetry/proto/trace/v1/trace.proto.

[32] Villamizar, M., Garcés, O., Castro, H., Verano, M., Salamanca, L., Casallas, R. and Gil, S., 2015. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. *2015 10th Computing Colombian Conference (10CCC)*. pp.583–590. Available from: https://doi.org/10.1109/ColumbianCC.2015.7333476.