


## Distributed Tracing for Troubleshooting of Native Cloud Applications via Rule-Induction Systems

**Arnak Poghosyan**

(Institute of Mathematics of NAS RA, Yerevan, Armenia  
VMware, Palo Alto, US

American University of Armenia, Yerevan, Armenia

 <https://orcid.org/0000-0002-6037-4851>


arnak@instmath.sci.am, apoghosyan@vmware.com, apoghosyan@aua.am)

**Ashot Harutyunyan**

(AI Lab at Yerevan State University, Yerevan, Armenia


Institute for Informatics and Automation Problems of NAS RA, Yerevan, Armenia

VMware, Palo Alto, US

 <https://orcid.org/0000-0003-2707-1039>, aharutyunyan@vmware.com)


**Naira Grigoryan**

(VMware, Palo Alto, US

 <https://orcid.org/0000-0003-3980-4500>, ngrigoryan@vmware.com)

**Clement Pang**

(VMware, Palo Alto, US

 <https://orcid.org/0000-0002-5821-0735>, clementp@gmail.com)

**Abstract:** Diagnosing IT issues is a challenging problem for large-scale distributed cloud environments due to complex and non-deterministic interrelations between the system components. Modern monitoring tools rely on AI-empowered data analytics for detection, root cause analysis, and rapid resolution of performance degradation. However, the successful adoption of AI solutions is anchored on trust. System administrators will not unthinkingly follow the recommendations without sufficient interpretability of solutions. Explainable AI is gaining popularity by enabling improved confidence and trust in intelligent solutions. For many industrial applications, explainable models with moderate accuracy are preferable to highly precise black-box ones. This paper shows the benefits of rule-induction classification methods, particularly RIPPER, for the root cause analysis of performance degradations. RIPPER reveals the causes of problems in a set of rules system administrators can use in remediation processes. Native cloud applications are based on the microservices architecture to consume the benefits of distributed computing. Monitoring such applications can be accomplished via distributed tracing, which inspects the passage of requests through different microservices. We discuss the application of rule-learning approaches to trace traffic passing through a malfunctioning microservice for the explanations of the problem. Experiments performed on datasets from cloud environments proved the applicability of such approaches and unveiled the benefits.

**Keywords:** cloud-native applications, application troubleshooting, distributed tracing, RED metrics, root cause analysis, explainable AI, rule-induction systems, RIPPER

**Categories:** I.2.6, I.5.4

**DOI:** 10.3897/jucs.112513

## 1 Introduction

Identification and remediation of the performance degradations of cloud applications require automated, real-time, and intelligent root cause analysis (RCA). Due to the complexity of microservices architecture (see [Knoche and Hasselbring, 2019, Lin et al., 2018, Cai et al., 2019, Liu et al., 2021, Tzanettis et al., 2022] with references therein), administrators are unable to perform timely detection and identification of IT issues. ML/AI empowered data analytics, or AI Ops (see [Notaro et al., 2020]), is designed to identify and resolve service incidents by supporting domain experts. RCA is one of the critical capabilities of AI Ops that will explain IT incidents in a human-readable medium through highly interpretable ML models. We show how rule-learning systems like RIPPER (see [Cohen, 1995]) can be the foundation of explainable RCA.

Application performance monitoring (APM) tools (see [Heger et al., 2017]) collect all available information for effectively addressing performance issues and managing applications (see [Harutyunyan et al., 2022, Harutyunyan et al., 2020a, Harutyunyan et al., 2020b, Poghosyan et al., 2016, Harutyunyan et al., 2018, Mahmud et al., 2021, Elsaadawy et al., 2019, Klaise et al., 2020, Vitali, 2022]). One of the main goals of APM is to enable the observability of cloud applications by aggregating data and outlining the system's overall health. Time series data, log data, and traces are the pillars of observability. This paper shows how explainable RCA can be built on top of application traces via rule-induction methods.

Distributed tracing (see [Heger et al., 2017, Parker et al., 2020] with references therein) is a modern technology for monitoring native cloud applications with microservices architecture. It is one of the best-known approaches for the monitoring of distributed systems. Traces observe end-to-end requests propagating through the distributed microservices and detect transaction slowdowns (see [VMware, 2022]). A single trace shows an individual request passage through the microservices. It contains a series of tagged time intervals known as spans (see Figure 1).

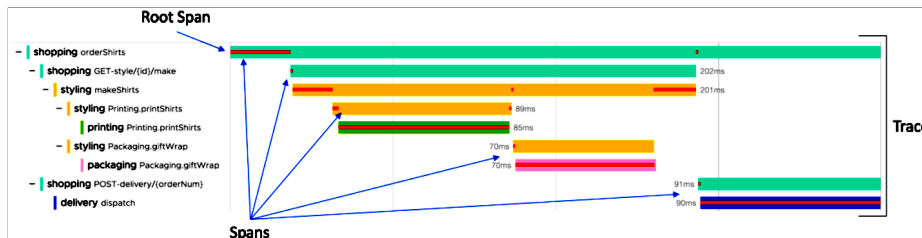


Figure 1: Individual trace of a simulated application ([https://docs.wavefront.com/trace\\_data\\_details.html](https://docs.wavefront.com/trace_data_details.html)).

A span contains metadata known as tags and application tags for better process resolution. Figure 2 shows the tags for the span "printShirts" from the service "printing." It shows the tags "cluster," "service," "location," "parent," "env", etc. One of the essential fields is the tag "error," which can be used to label traces. This field's value="true" indicates that the corresponding trace is erroneous. Otherwise, a trace is expected if the field is missing (typical). We can also label traces based on other characteristics.

The troubleshooting of an application in case of some issues can be manually performed via traces browser (see Figure 3), which shows a group of traces corresponding

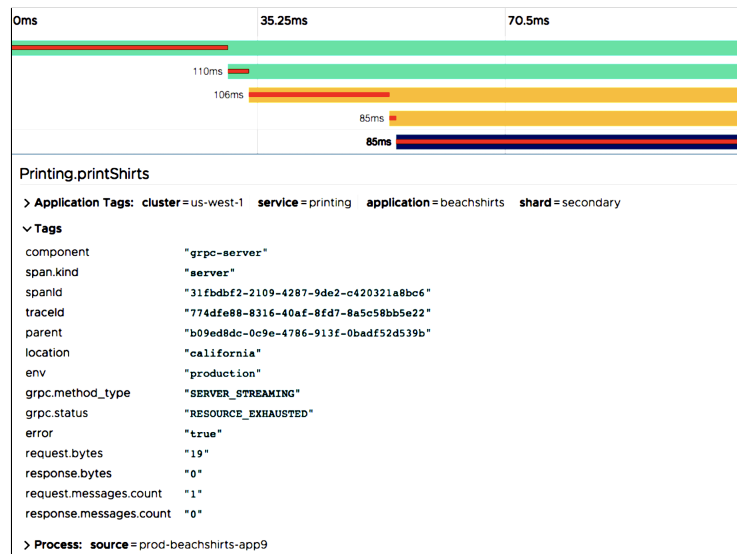


Figure 2: The tags of a span of a simulated application ([https://docs.wavefront.com/tracing\\_traces\\_browser.html](https://docs.wavefront.com/tracing_traces_browser.html)).

to a service, unveils their durations, indicates anomaly traces, and for each trace presents the structure of spans.

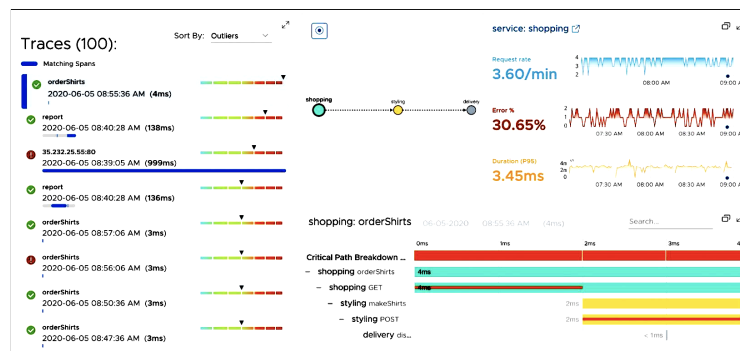


Figure 3: Traces browser for a simulated application ([https://docs.wavefront.com/tracing\\_traces\\_browser.html](https://docs.wavefront.com/tracing_traces_browser.html)).

The health of a service can be tracked by the values of RED (rate, error, duration) metrics corresponding to the number of requests per minute, the number of errors (failed requests per minute), and the p95 quantile of trace durations in a minute. The change in behaviors of RED metrics can indicate some issues.

Moreover, some hard thresholds can be put on top of those metrics, which violations will trigger alerts indicating users to start the process of troubleshooting for the root cause

analysis of issues. Unfortunately, all those tasks can be performed only for a limited number of services and traces. Universal manual troubleshooting for all available services and traces is not feasible due to the complexity of interrelations between application components and a large volume of trace information. We consider a more general and automated approach to the problem of application troubleshooting based on trace traffic.

Trace traffic shows how applications and services interact. Application map is the visualization of the tracing traffic (see Figure 4). The colors of services indicate the statuses of the corresponding microservices. The colors optionally can be assigned by the values of RED metrics. The red-colored microservices have poor performances, and our main goal is to understand/explain those problems. We can select a poor-behaving microservice, collect tracing traffic passing through it, and analyze via explainable AI (XAI) methods for some acceptable interpretations of the performance degradations in trace types, spans, and tags.

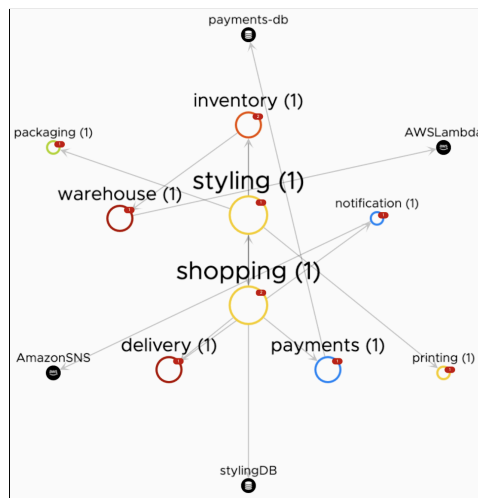


Figure 4: Application map for a simulated application ([https://docs.wavefront.com/tracing\\_ui\\_overview.html](https://docs.wavefront.com/tracing_ui_overview.html)).

## 2 Related Work

System administrators can no longer perform real-time decision-making due to the growth of large-scale distributed cloud environments with complicated, invisible underlying processes. Those systems require more advanced and ML/AI-empowered intelligent RCA with explainable and actionable recommendations (see [Solé et al., 2017, Harutyunyan et al., 2019, Poghosyan et al., 2021a, Poghosyan et al., 2016, Poghosyan et al., 2021b, Marvasti et al., 2013] with references therein). XAI (see [Barredo Arrieta et al., 2020]) builds user and AI trust, increases solutions' satisfaction, and leads to more actionable and robust prediction and root cause analysis models. Many users think it is risky to trust and follow AI recommendations and predictions blindly, and they need to understand the foundation of those insights. Many ML approaches like decision trees and rule-induction

systems (see [Fürnkranz et al., 2012]) have sufficient explainability capabilities for RCA. They can detect and predict performance degradations and identify the most critical features (processes) potentially responsible for the malfunctioning. In many applications, explainable outcomes can be more valuable than conclusions based on more powerful approaches that act like black boxes. Rule learners are the best if outcomes' simplicity and human interpretability are superior to the predictive power (see [Fürnkranz et al., 2012]). The list of known rule learners consists of many exciting approaches like association rules, APRIORI, CN2, AQ, FOIL, STUCCO, OPUS, RIPPER, and many others. We refer to [Fürnkranz et al., 2012] for a more detailed description of available algorithms, their comparisons, and historical analysis. It contains relatively rich references and describes several applications.

Data overfitting is a well-known issue that also affects rule learning systems. RIPPER (see [Cohen, 1995]) was the first rule-learning system that effectively countered the overfitting problem (see [Fürnkranz and Kliegr, 2015, Fürnkranz, 1997]). It is based on several previous works, most notably the incrementally reduced error pruning (IREP) idea described in [Fürnkranz and Widmer, 1994]. In several independent studies, RIPPER has proved to be among the most competitive rule-learning algorithms available today. It is competitive with C4.5RULES (see [Quinlan, 2014]) without losing IREP's efficiency. RIPPER is a classification rule induction approach, and the crucial steps for its realization are data labeling and feature selection/extraction in the case of massive datasets. Many authors have tried to improve it. An interesting approach is FURIA (see [Hühn and Hüllermeier, 2009]).

Applying rule learning algorithms to industrial and IT problems has a long history. Papers [Suriadi et al., 2013, Lin et al., 2020] consider analysis based on log data. Paper [Suriadi et al., 2013] proposes an approach to enrich and transform process-based logs for RCA by applying classification techniques. The idea is to transform event logs by using classical data mining techniques. The final classification-ready problem was successfully treated by J48 and JRip, available in the machine learning library Weka. Algorithm J48 is the Weka implementation of the well-known C4.5RULES learning algorithm (see [Quinlan, 2014]). Its improvement is known as algorithm C5.0. Algorithm JRip is the Weka implementation of RIPPER. It resulted in much simpler rules with comparable accuracy of classification. Paper [Lin et al., 2020] considers the problem of RCA in a large-scale production environment based on structured logs. The challenge is the complexity of services running across global data centers. The authors explore the application of the Apriori algorithm (see [Agrawal et al., 1993]) with subsequent improvement by the FP-Growth approach (see [Han et al., 2004]).

Papers [Lee and Stolfo, 1998, Helmer et al., 1998, Helmer et al., 2002] consider the problem of intrusion and abuse detection in computer systems in networked environments. Their method is based on the system calls executed by a program like primitive traces. Paper [Lee and Stolfo, 1998] experimented with "sendmail" system call and network "tcpdump" data. In the first example, each trace data file has two columns of integers. The first is the process IDs, and the second is the system call names. The set includes known normal and abnormal traces. Then, they apply RIPPER for a rule induction in a specific manner - each record has the same number of positional attributes plus a label. The final rules were used to predict whether a sequence is abnormal or normal. Experiments showed that the expected behavior of a program execution could be established and used to detect its anomaly states. RIPPER was also applied to "tcpdump" data for anomaly predictions. Furthermore, the authors combined different classifiers to improve the effectiveness of detecting intrusions. The list included association rules and frequent episodes algorithm (see [Mannila et al., 1995]) with promising results.

Paper [Helmer et al., 1998] continued the previous work by proposing a feature vector construction technique for system call traces. The traces were encoded as binary-valued bits in feature vectors. Each bit in the vector indicates whether a known system call sequence appeared during the execution of a process. Those feature vectors were used for rule induction via RIPPER.

Paper [Helmer et al., 2002] went further and discussed an interesting approach for complexity reduction of learning algorithms. Naturally, the complexity is directly connected with the number of features involved in the learning. It should be a very reasonable application of learning systems to important features. Feature subset selection has been shown to improve the performance of a learning algorithm and reduce the effort and amount of data required for machine learning on a broad range of problems (see [Liu and Motoda, 1998]). Paper [Helmer et al., 2002] considered genetic algorithms for feature subset selection (see also [John et al., 1994]). The main open issue of those papers is the application of the methods to heterogeneous distributed systems. The current paper addresses this problem and describes a far more general approach to feature construction and data labeling with different levels of resolution.

The current paper is the extension of our paper published in the 3rd CODASSCA Workshop on Collaborative Technologies and Data Science in Artificial Intelligence Applications, Yerevan 2022, Armenia (see [Poghosyan et al., 2022]). Several US patents are available regarding the ideas of this paper (see [Nag et al., 2021, Poghosyan et al., 2021c]).

### 3 Methodology

RCA's main goal is to understand a problem's root causes to identify appropriate resolution procedures. Even if the root causes are not directly visible, RCA must provide sufficient explainability of a problem for the acceleration of its remediation or to prevent future occurrences. It means that the capabilities of detection or prediction of performance degradations need to be improved for ML-empowered RCA solutions. A sufficient level of explainability is the main requirement from those ML solutions for spreading light onto the underlying complex processes.

Fortunately, there are many powerful learning approaches with a high level of explainability, like decision trees and rule-induction methods. In this paper, we illustrate the power of RIPPER (see [Cohen, 1995]) to explain the performance degradations of application microservices monitored via distributed tracing. According to [Fürnkranz and Kliegr, 2015], RIPPER is still state-of-the-art in inductive rule learning. It has some important technical characteristics, like supporting missing values, numerical and categorical variables, and multiple classes. According to [Cohen, 1995], it scales nearly linearly with the number of instances in large datasets.

We experimented with Weka's RIPPER implementation, or JRip (see [Witten et al., 2005]). This implementation is very stable and fast. It takes seconds (up to a minute) to induce rules for a dataset with thousands of traces. The default algorithm is known as RIPPER2 (see [Cohen, 1995]). JRip has a more general implementation known as RIPPER $k$  that repeatedly optimizes  $k$  times. We tried RIPPER5 and RIPPER10, which sometimes resulted in fewer rules but took longer for the execution.

RIPPER aims to find regularities in data in the form of an IF-THEN rule (see [Fürnkranz and Kliegr, 2015]). The condition of a rule (body) is composed of a conjunction of Boolean terms, each consisting of a constraint that needs to be satisfied by a trace. The rule is said to fire if all constraints are satisfied, and a trace is said to be covered by

the rule. The rule head is a class label predicted in case the rule fires. The RIPPER rules can be simple (with only one condition) or complex (consisting of multiple constraints). The importance of a rule can be characterized by the following well-known measures (see [Fürnkranz and Kliegr, 2015]):

$$Confidence = N(body \rightarrow head) / N(body),$$

and

$$Coverage = N(body \rightarrow head) / N(positive\ class),$$

where  $N(body \rightarrow head)$  is the number of traces that satisfy both the body and head of the rule (correct classifications by the corresponding rule),  $N(body)$  is the number of traces that satisfy the body of the rule (both correct and incorrect classifications by the rule) and  $N(positiveclass)$  is the number of traces labeled as “interesting for explanations.” We can set some thresholds for acceptable rules. Say, the coverage and confidence of the rules should be greater than 20% and 80%, respectively.

We recommended a slightly modified procedure for retrieving rules compared to the classical direct application of RIPPER. For many problems, several spans or tag values can equally explain the root cause. RIPPER will randomly select some of them for rule retrieval. However, not all acceptable rules will be equally relevant for system administrators for further remediation progress. We apply RIPPER in cycles (iterations) to show all hidden recommendations. This is somewhat similar to a feature extraction process but with extra caution, as we don’t know which attributes will lead to actionable recommendations without application domain knowledge. After each iteration of rule extraction, all features appearing in the previous rules should be removed from the dataset, and the rule-learning algorithm should be applied again. We iterate the procedure until some stopping criteria – time limitation or the weakness of the remaining rules.

## 4 Data Preparation and Algorithms

The analysis of tracing traffic passing through a specific microservice starts with data preprocessing. The traffic can contain hundreds or thousands of traces with different numbers of spans and tags, which are application-specific. Figure 5 shows a portion of trace traffic in a semi-structured format. It contains the names of traces as trace-IDs, then after each trace, the list of spans with process names, and after each span, the corresponding tags. Algorithm RIPPER requires tabular data (like dataframes in Pandas library). During the transformation, we remove some of the fields containing redundant information, useless for the insights of RCA, like “traceID,” “spanID,” “startMs,” and many others. It should be very reasonable to denoise trace traffic based on expert knowledge or user feedback.

We consider three approaches for data tabularization, which lead to three algorithms with different levels of resolution and complexity. The first one (Algorithm A) works on a span level. We form the list of all distinct span names existing in trace traffic and put them as the names of the columns of a dataframe. Then, for each trace (as a row), we verify the names of its spans, and in the corresponding columns, enter value = 1. The other columns contain missing values as that specific trace doesn’t contain those spans. The entire dataframe consists of ones and missing values. Fortunately, RIPPER ignores those missing values and explains the output based on the existing spans in a trace. Eventually, the number of rows in the dataframe coincides with the number of

```
{'traceId': '6a538fb0-6a6f-4d5a-bbe7-9fd7295f939e',
 'spans': [{'name': 'ctp.auth.jdbc-execute',
            'host': 'a-at11auth14',
            'startMs': 1597299097216,
            'durationMs': 1,
            'spanId': 'e8df55f9-648b-4d38-9e25-c634daf2c3e9',
            'traceId': '6a538fb0-6a6f-4d5a-bbe7-9fd7295f939e',
            'annotations': [{'parent': '76014304-3e43-47f5-a7be-b662a430afb0'},
                            {'followsFrom': '76014304-3e43-47f5-a7be-b662a430afb0'},
                            {'span.kind': 'client'},
                            {'component': 'java-jdbc'},
                            {'service': 'auth'},
                            {'application': 'ctp'},
                            {'shard': 'at11'},
                            {'cluster': 'agile'}],
```

Figure 5: A portion of a trace-traffic in a semi-structured form.

traces in trace traffic, and the number of columns coincides with the number of distinct spans (processes).

Trace labeling can be performed via the tag "error," where the value="true" indicates that the corresponding trace is erroneous. A missing tag "error" indicates that the trace is normal. We can keep the number of erroneous traces slightly smaller than normal traces. In that case, RIPPER will assume that the class of erroneous traces is positive and will extract rules (containing only span names) for their explanations.

It can be useful also to construct a column in a dataframe that will indicate the type of a trace. Different ideas are known for trace-type determination. One of the ideas (see [Nag et al., 2021]) is to apply grouping of the traces based on the similarity of the sets of spans. Then, each group will define a trace type. However, technically, it is difficult to accomplish. The most straightforward idea is to use the root span available for some traces. All traces without root spans can be grouped as a single type. Hence, Algorithm A deals with spans and trace types to explain the set of erroneous traces. Sometimes, this level of resolution should be sufficient for problem identification.

The second approach (Algorithm B) works on a tag level. The process is similar to the previous one by using the list of span names in combination with tag names as the names of columns in the corresponding dataframe. Trace type and output columns can be copy-pasted from the previous construction. The dataframe cells contain value=1 for the corresponding spans, value="NaN" for the missing ones, or the corresponding values of tags. This dataframe is much bigger than the first one, and Algorithm B is more complex than Algorithm A. RIPPER rules will contain the tags trying to explain the output by the values of tags. Hence, this approach has better resolution, and the corresponding insights may be more helpful.

The third approach (Algorithm C) uses a combined dataframe of the first two. This will be the most complex but the most complete approach. RIPPER rules will use both span names and tag names with their values, trying to find the best explanations.

RIPPER is relatively efficient in the sense of execution time. However, the number of spans and tags heavily affects the complexity. Those numbers are application-specific. In the case of hundreds of distinct spans, the execution of RIPPER can take several seconds up to a minute. This time range is connected with the noise in data. It is well-known that RIPPER is time-consuming for noisy datasets. In the case of thousands of distinct



tags, the execution of RIPPER will take several minutes. The latest can be a problem in real-time monitoring systems when users are very demanding for fast executions. In those extreme situations, we can preprocess data for Algorithm A and show results regarding span names. Before a user can inspect the insights for remediation actions, we can proceed with Algorithms B or C for better problem explanations. In all cases, it is essential to measure the coverage and confidence of rules and limit the number of insights for users.

## 5 Experimental Results and Discussions

We perform experiments with data from actual cloud environments. Due to confidentiality, we will hide their names and use Customer I, II, and III. In all scenarios, we selected suspicious microservices from the corresponding application maps and collected some traces from the trace traffic passing through those components.

We aim to detect those spans, tags, and tag values that can explain the services' troubles. We show and discuss the outcomes of Algorithms A, B, and C.

### 5.1 Customer I

We selected 5428 traces for the first customer (Customer I). Algorithm A requires trace types and the distinct names of spans. Preliminary analysis showed 8 different trace types and 30 distinct spans. Hence, Algorithm A will work with the dataframe with 31 columns and 5428 rows. The first column contains information regarding the trace type. It is a categorical feature with 8 classes corresponding to different trace types. The remaining 30 columns contain values "1" or "NaN." RIPPER ignores the missing values and explains the output based on the existing spans.

We performed data labeling based on erroneous traces. Our example contains 2393 erroneous traces and 3035 normal ones. We see that the number of erroneous traces is smaller than that of normal ones. RIPPER assumes that the smaller class is positive and tries to find the rules for explaining the erroneous traces. Algorithm A will solve that problem based on types and span names. Figure 6 shows the outcome of RIPPER applied to the described dataframe (Algorithm A).

```

JRIP rules:
=====
(type = ctp.prov-repl.call-remedy) and
(span = ctp.remedy-webapp.POST/arsys/services/ARService)
=> trace = erroneous (2343/1087)

```

Figure 6: The outcome of Algorithm A for the dataset of Customer I.

It shows a complex rule as the combination of two conditions. The first condition is the existence of type "ctp.prov-repl.call-remedy". The errors can be connected with the traces with the specified root span. The second condition shows the existence of the span "ctp.remedy-webapp...ARService" in traces. In combination, it means that all traces

with the specified root span and containing the span "ctp.remedy-webapp...ARService" explain a portion of errors in the trace traffic.

The fraction at the rule's end will help calculate the corresponding scores. The numerator of the fraction 2343 shows how many times the rule has been fired. The denominator of the fraction 1087 shows the number of misclassifications. The rule has been fired for normal traces 1087 times. The coverage is 52% (see Table 1).

The Rules of Figure 6	Coverage	Confidence
Rule #1	0.52	0.54

Table 1: The coverage and confidence of the rules of Figure 6.

The confidence (the number of correct classifications divided by the number of fired rules) is 54% (see Table 1). The execution time of Algorithm A was less than a second. Unfortunately, the value of confidence is dissatisfactory. We will not show the rule to a user. We cannot explain the set of erroneous traces by the types or spans of traces with sufficient accuracy.

Let us continue with Algorithm B, which will explain the erroneous traces via tags. We found 489 distinct tags corresponding to different spans by storing them as span names plus tag names. The corresponding dataframe contains 489 columns and 5428 rows. The labels are the same. We can include or not include the column "type." Figure 7 shows the outcome of Algorithm B. We see two simple rules with perfect confidence 100%.

```

JRIP rules:
=====
ctp.prov-repl.call-remedy_annotations__spanLogs = "true"
=> trace = erroneous (2276/0)

ctp.remedy-webapp.POST/arsys/services/ARService_annotations_http.status_code = 500
=> trace = erroneous (114/0)
    
```

Figure 7: The outcome of Algorithm B for the dataset of Customer I.

The first rule refers to the span "ctp.prov-repl.call-remedy" which was shown previously as the outcome of Algorithm A. Algorithm B reveals one of its important tags "\_annotations\_\_spanLogs" with the corresponding value "true." The coverage of this rule is 95% (see Table 2).

The Rules of Figure 7	Coverage	Confidence
Rule #1	0.95	1
Rule #2	0.05	1

Table 2: The coverage and confidence of the rules of Figure 7.

The coverage of the second rule is around 5% (see Table 2). JRip also listed other rules with even smaller coverage. We are not showing them. The first rule has ideal coverage and confidence scores for sending to an end-user to explain the microservice performance degradation. The second rule should be excluded due to its small coverage.

It is possible that the revealed rules are not helpful in problem resolution. Although they are perfectly correlated with the errors, direct programming may have resulted. For example, the developers of applications are adding the tag "error" with the value "true" for a span if its "status\_code" is "500" or the value of "\_spanLogs" is "true." That is why the confidence of those rules is very high.

We recommend the application of RIPPER in iterations by removing the spans and tags that appeared in the previous rule from the corresponding dataframes and reapplying the algorithm. We may continue until some stopping criteria are accomplished. One of the restrictions can be the time of executions. The second criterion can be the small values for coverage or confidence of the previous rules.

Returning to the previous example, we remove two features in the rules of Figure 7. The following Figure 8 shows the rules after the reiteration of the procedure. Information contained in those rules should be more helpful.

```

JRIP rules:
=====
ctp.prov-repl.call-remedy_annotations_peer.address = https://agile-remedymid... = Devices
=> trace = erroneous (1766/9)

(ctp.prov-repl.call-remedy_annotations_peer.address = https://agile-remedymid... = Contacts) and
(ctp.prov-repl.call-remedy_annotations_soap.action = urn:Contacts/CTPEngineeringContactSet)
=> trace = erroneous (88/0)

```

Figure 8: The outcome of Algorithm B for the dataset of Customer 1 after the second iteration.

They are showing some specific peer addresses somehow related to the errors. The first rule covers 73% and confidence 99.5% (see Table 3). The second rule has a small coverage but 100% confidence (see Table 3). We can send the first rule to a user to validate the insight.

The Rules of Figure 8	Coverage	Confidence
Rule #1	0.73	0.995
Rule #2	0.04	1

Table 3: The coverage and confidence of the rules of Figure 8.

It is reasonable to start with Algorithm A. It will take several seconds to return the insights. Then, we can continue with Algorithms B and C while a user inspects the first set of rules. This will decrease the waiting time for a user as a sensitive constraint.

## 5.2 Customer II

In this case, we selected 11894 traces from the trace traffic passing through a malfunctioning microservice. The number of distinct trace types is 99, much bigger than in the previous example. Similarly, the number of distinct spans is also very big. We found 186 such spans. Algorithm A will work with a dataframe composed of 187 columns and 11894 rows. We performed data labeling via erroneous traces. We detected 2020 erroneous traces and 9873 normal ones. The execution time of Algorithm A is up to a second with JRip implementation. Figure 9 shows the first three rules.

```

JRIP rules:
=====
span = SreHealth.vpxd.vim.SessionManager.logout
=> trace = erroneous (868/17)

span = SreHealth.vpxd.vim.option.OptionManager.queryView
=> trace = erroneous (501/0)

span = SreHealth.vpxd.vim.ServiceInstance.currentTime
=> trace = erroneous (511/211)

```

Figure 9: The outcome of Algorithm A for the dataset of Customer II.

The first rule has 42% coverage and 98% confidence (see Table 4). It recommends the span "...SessionManager.logout". The second rule covers 25% and confidence 100% (see Table 4). It recommends "...OptionManager.queryView". The last rule has unacceptable 59% confidence (see Table 4). We will recommend only the first two rules for further consumption.

The Rules of Figure 9	Coverage	Confidence
Rule #1	0.42	0.98
Rule #2	0.25	1
Rule #3	0.15	0.59

Table 4: The coverage and confidence of the rules of Figure 9.

We continue with the second iteration after removing all features that appeared in the first iteration. Figure 10 reveals the new set of rules.

The first rule has ideal confidence and 25% coverage (see Table 5). We can add it to the previous list of recommendations. We can skip the remaining rules due to the small coverage (see Table 5).

We can continue with several iterations for each rule, calculating the scores and selecting the ones with sufficient coverage and confidence. Then, we can sort the rules by confidence in decreasing order and reveal the top five recommendations to a user. As mentioned above, the list of equivalent recommendations can be long. The best solution is to incorporate user feedback in the process of dataframe construction. Users can outline those spans or tags that can be helpful to see in the insights. It can help to decrease the number of columns of the dataframe and optimize the resource consumption by JRip.

```

JRIP rules:
=====
span = SreHealth.vpxd.vim.option.OptionManager.queryView.Iro
=> trace = erroneous (500/0)

span = SreHealth.analytics.logout
=> trace = erroneous (213/21)

span = SreHealth.vpxd.vmodl.query.PropertyCollector.retrievePropertiesEx
=> trace = erroneous (185/25)

span = SreHealth.analytics.loginByToken
=> trace = erroneous (97/0)

```

Figure 10: The outcome of Algorithm A for the dataset of Customer II after the second iteration.

The Rules of Figure 10	Coverage	Confidence
Rule #1	0.25	1
Rule #2	0.1	0.9
Rule #3	0.08	0.86
Rule #4	0.05	1

Table 5: The coverage and confidence of the rules of Figure 10.

Now, let us return to Algorithm B. The number of distinct tags is enormous for the dataframe of Customer II. We found such 3002 distinct tags. The labels are the same as for Algorithm A. The execution time of JRip is around 20 seconds. Figure 11 shows the outcome of Algorithm B.

```

JRIP rules:
=====
SreHealth.vpxd.vim.SessionManager.logout_annotations_error.type = Vim::Fault::NotAuthenticated
=> trace = erroneous (851/0)

SreHealth.vpxd.vim.option.OptionManager.queryView_host=sddc-prd.209ad895-....
=> trace = erroneous (501/0)

SreHealth.vpxd.vim.ServiceInstance.currentTime_annotations_error.type = Vim::Fault::NotAuthenticated
=> trace = erroneous (297/0)

SreHealth.vpxd.vmodl.query.Iro_annotations_session = 52ce5ab9-beef-bbc8...
=> trace = erroneous (160/0)

SreHealth.vpxd.vim.view.ViewManager_annotations_error.type = Vim::Fault::NotAuthenticated
=> trace = erroneous (123/0)

```

Figure 11: The outcome of Algorithm B for the dataset of Customer II.

We see interesting rules with 100% confidence (see Table 6). Three (the first, third, and fifth) directly indicate the same "error.type" = "NotAuthenticated". The second

indicates the specific host and the fourth refers to the session.

The Rules of Figure 11	Coverage	Confidence
Rule #1	0.42	1
Rule #2	0.25	1
Rule #3	0.15	1
Rule #4	0.08	1
Rule #5	0.06	1

Table 6: The coverage and confidence of the rules of Figure 11.

This example leads to another interesting data labeling idea. What if we want to explain the root cause of a specific error type? We can restart the labeling according to that specific value. For the example outlined in Figure 11, the label will take value 1 if "error.type" = "Vim::Fault::NotAuthenticated", and value 0, otherwise. For this specific example, we found 851 erroneous traces corresponding to that "error.type" and applied Algorithm C for details. Preliminarily, we removed all fields containing "status.codes" and "logouts" as the previous rules already indicated the problem description in those terms. Figure 12 explains the source of that error. It indicates that almost all errors are connected with a specific host "sddc-prd.209ad...".

<p><b>JRIP rules:</b>            =====</p> <p>SreHealth.vpxd.vim.SessionManager.logout_host = "sddc-prd.209ad895-..."</p> <p>=&gt; error.type = "Vim::Fault::NotAuthenticated" (868/17)</p>
---

Figure 12: The outcome of Algorithm C for the dataset of Customer II, where data labeling was performed via "error.type" = "Vim::Fault::NotAuthenticated" values.

It is a valuable insight for a system administrator, leading to the direct source of the problem. A more general multi-class classification problem can be solved by putting different labels on all available values for a specified tag. The RCA will explain the origin of traces with the specific tag values via trace types, spans, or other tag values in a dataset related to a problem. Fortunately, RIPPER is dealing with multi-class problems with the same efficiency.

## 6 Customer III

We selected 5899 traces from the corresponding trace traffic and detected 26 distinct trace types. We also found 451 distinct spans and 4465 distinct tags. The number of distinct spans was rather huge. We verified the potential of Algorithm C for this scenario. The corresponding dataframe had 4917 columns and 5899 rows. We performed labeling based

on the errors. We found 2754 erroneous and 3145 normal traces. The execution time of Algorithm C was 15 seconds. Actually, quite acceptable even for this big dataframe.

In general, the execution time depends on the number of distinct spans and tags. However, worth noting that it also depends on the number of classes in the categorical variables (the distinct values of tags). A bigger number of classes leads to longer execution times. Another important influential characteristic is noise. It is well known that RIPPER needs more time to construct rules in case of noisy datasets.

Figure 13 reveals that the most important component for explaining the errors of traces is the type (root span of a trace).

```

JRIP rules:
=====|
type = Zipkin.ad-selector-canary.user-db-with-retry.getuserinfostruct
=> trace = erroneous (1317/0)

type = Zipkin.ad-selector-canary.user-db.getuserinfostruct
=> trace = erroneous (1321/0)

```

Figure 13: The outcome of Algorithm C for the dataset of Customer III.

It shows two important types "Zipkin.ad-selector-canary.user-db-retry.getuserinfostruct" and "Zipkin.ad-selector-canary.user-db.getuserinfostruct". Both processes are connected to a database. It means that the performance degradation of a microservice is connected with the DB. Table 7 presents the corresponding coverage and confidence of the rules.

The Rules of Figure 13	Coverage	Confidence
Rule #1	0.48	1
Rule #2	0.48	1

Table 7: The coverage and confidence of the rules of Figure 13.

There is an interesting modification of algorithms connected with spans. The idea is to modify the corresponding dataframes by considering the number of repetitions of the same spans in a trace. Until now, we have counted each span only once, no matter how many times a specific span appeared in the same trace. The entries of the dataframe were "1" or "NaN." Now, the entries will be "the number of appearances" or "NaN." This modification can tremendously strengthen the algorithms as erroneous micro-processes have a habit of repetitions.

For example, assume a problem with a credit card payment. In case of some malfunctioning, a user will repeat the payment, hoping to pay successfully, and the span corresponding to the payment will appear several times. This powerful modification is relatively simple to show for Customer III. Application of Algorithm A returns an empty set of rules, meaning explaining the erroneous traces via spans is impossible. However, the modification of Algorithm A reveals some insights shown in Figure 14.

JRIP rules: =====
(Zipkin.ad-selector-canary.user-db.getuserinfostruct = 2) and (Zipkin.ad-selector-canary.user-db-with-retry.getuserinfostruct = 1) => trace = erroneous (2680/0)
(Zipkin.ad-selector-canary.infer-scores = 1) and (Zipkin.ad-selector-canary.inference.selectandpredict = 1) => trace = erroneous (20/0)

Figure 14: Modification of Algorithm A by incorporation of span-repetitions in a trace for Customer III.

The first rule has a large coverage and 100% confidence (see Table 8). Interestingly, it is similar to the insights of the previous figure.

The Rules of Figure 14	Coverage	Confidence
Rule #1	0.97	1
Rule #2	0.007	1

Table 8: The coverage and confidence of the rules of Figure 14.

## 7 Trace-Latency Based RCA

Labeling traces based on an "error" tag is a natural possibility, resulting in a powerful explanation engine. Another essential identifier of service performance degradation is the traces' duration (latency). Usually, the execution time of a service has some average duration. Very short (usually with a zero duration) or long traces/spans indicate a shift from the normality and should be explained.

We outline two different approaches for the labeling of traces via durations. First, we need to separate traces into types (specific processes) and collect examples from different groups. If the number of examples in each group is rather significant, then  $p_{05}$  and  $p_{95}$  quantiles (or any other reasonable ones) of each group can serve as thresholds for the detection of small and large latencies. All traces in a group with durations smaller than  $p_{05}$  will get label  $-1$ , with durations longer than  $p_{95}$  will get label  $1$ , otherwise label  $0$  (as normal traces). Then, RIPPER will explain separately both abnormal latency groups of traces. Those calculations can be made very efficient using the t-digest approach (see [Dunning and Ertl, 2019]), which can accurately estimate extreme quantiles via a streaming approach based on traces collected over several days or months.

If the number of samples in each group is small, then the suggested method will always detect some false positives. In that case, we need a simple outlier detection method. For example, we can apply the whiskers' method, which defines thresholds by the formulae

$$upper = q_{75} + 1.5 * (q_{75} - q_{25}),$$



and

$$lower = q_{25} - 1.5 * (q_{75} - q_{25}).$$

Figure 15 shows how the whiskers' method can be used for an outlier detection for a specific trace type.

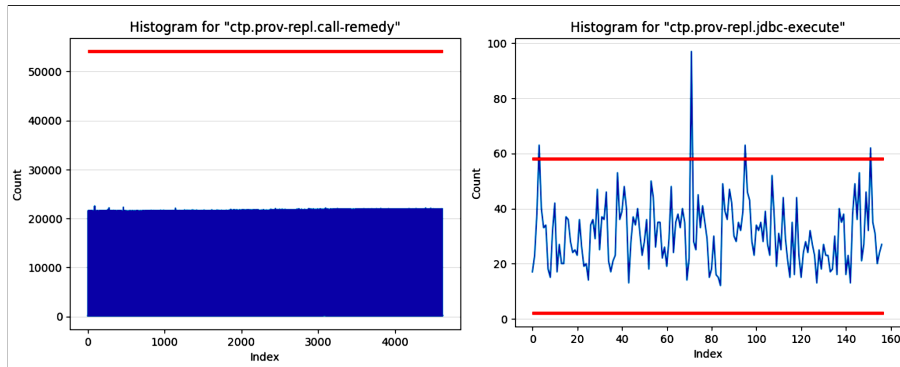


Figure 15: The whiskers' method for outlier detection in trace types.

Red lines correspond to the "upper" and "lower" thresholds calculated for each group. Each figure shows the distribution of trace durations (titles of images indicate the root spans). The left figure shows relatively compact durations of traces in that group, and the whiskers' method will not detect outliers. Some traces on the right figure violate the "upper" threshold.

Algorithms A, B, and C will work precisely similarly. The only difference is in labels; instead of errors, now we are explaining the durations of processes. Let us consider a dataset from the environment of Customer II. Labeling traces based on latency has resulted in 31 low-duration, 1127 high-duration, and 10735 normal traces. Figure 16 explains those outlying traces.

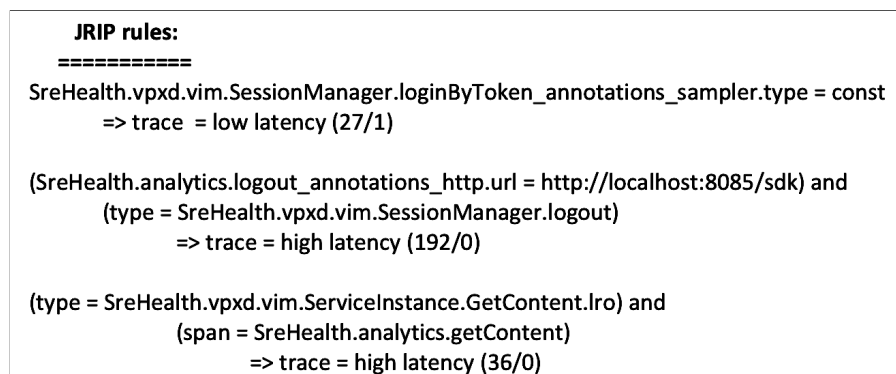


Figure 16: Latency-based RCA for Customer II via Algorithm C.

We removed some of the poor rules with small coverage and confidence. The first rule in Figure 16 explains the low-duration traces. It correctly explains 26 occurrences from 31 erroneous traces. The second two rules describe high-duration traces. They have low coverage but perfect confidence (see Table 9).

The Rules of Figure 16	Coverage	Confidence
Rule #1	0.84	0.96
Rule #2	0.17	1
Rule #3	0.03	1

Table 9: The coverage and confidence of the rules of Figure 16.

Now, let us consider a dataset from the environment of Customer III. As we mentioned before, we detected 8 different trace types. For each of them, we performed outlier detection and found 3027 erroneous traces with high latency (the label is "1"), 16 erroneous traces with low latency (the label is "-1"), and 6957 normal ones (the label is "0").

The rules of Figure 17 explain only the traces with high latency. RIPPER didn't find patterns for the low-latency traces. We applied modified Algorithm A, which uses the number of appearances of spans in traces. Some traces in trace traffic have a root span. The root span defines the type of trace. However, some traces are arriving without the root span. We are collecting them in a unified group named "root\_absent." We can remove previously appeared spans and tags and reiterate the procedure for more useful insights.

```
(Zipkin.graphql.styling_service.get_profile_structured_styles = 1) and
(type = root_absent)
=> trace = high latency (1450/221)

(Zipkin.r2.cassandra_annotations_column_family = LastModified) and
(Zipkin.preference_service.cassandra.execute = 1) and
(Zipkin.r2.award_service.get_awardings_by_identifiers = 2) and
(type = root_absent)
=> trace = high latency (473/32)
```

Figure 17: Latency-based RCA analysis for Customer III via modified Algorithm A.

Table 10 presents the coverage and confidence of the rule in Figure 17. The confidences of the rules are acceptable. The coverage of the first rule is also acceptable.

## 8 Filtering of RCA Recommendations via Baseline Estimation

Validation of recommendations before assigning them to an IT specialist for further remediation actions is an important milestone. We have set up several importance scores

The Rules of Figure 17	Coverage	Confidence
Rule #1	0.4	0.85
Rule #2	0.15	0.93

Table 10: The coverage and confidence of the rules of Figure 17.

for the prioritization of rules. However, there can be more general sources of noise that are worth further consideration. Many applications carry background noise (a bunch of erroneous traces), which does not impact the performance. The corresponding trace traffic contains a bucket of erroneous traces, and RIPPER can explain them with relevant rules. However, those rules are misleading for the final goal of recovering the performance degradation. More valuable insights will be recovered from those erroneous traces resulting from a change in trace traffic. Here, we suggest a procedure that can help to filter out the recommendations corresponding to an application background noise.

Change detection is possible while comparing two portions of trace traffic. The first one corresponds to a time window without any performance issues. We call it as peacetime period. Worth noting once more that RIPPER applied to this trace traffic may result in several strong rules that we want to eliminate from the final list of recommendations. The second one corresponds to a period with some IT issues. We call it as wartime period. We want the explanation of the erroneous traces in the wartime period subject to the peacetime one (conditional RCA). We call this approach war-peace-time RCA.

A naïve approach to war-peace-time RCA would assume the application of JRip separately to both dataframes with a filtering procedure for the wartime rules subject to the peacetime rules. For example, we can exclude those spans and tags from the wartime rules that simultaneously appear in the peacetime rules. However, we don't have a consistent procedure for this task. We suggest another automatic procedure that directly retrieves the required rules, excluding the process of manual filtering.

The first step is the application of RIPPER to the peacetime period. We need only the trained classification model without rules. The second step is applying the trained model to the wartime period. We are interested only in the misclassified traces we use for trace relabelling. All erroneous traces in the wartime period that were classified correctly will change their labels from "1" (default label for the erroneous traces) to "0". We are not interested in the correctly classified erroneous traces, as they probably already appeared in the peacetime dataset. We keep only those labels that the peacetime model misclassified. Finally, we apply RIPPER to the relabelled wartime dataframe and derive the required rules. This procedure can be used for three Algorithms: A, B, and C. The entire procedure can be reiterated as before.

Let us consider war-peace-time RCA for a dataset from the Customer I environment. Algorithm B applied to a peacetime period revealed the importance of the tag "http.status\_code = 500" associated with the span "ctp.remedywebapp.POST/arsys/services-/ARService". We have already seen this recommendation before. The rule has 45.4% coverage and 100% confidence. Application of RIPPER to a wartime period revealed the same recommendation with 100% confidence and 49.6% coverage. However, the war-peace-time RCA returns an empty list of recommendations after filtering some weak rules. It means that war-peace-time RCA is capable of natural filtering of redundant rules. The main drawback of this approach is the difficulty of selecting the corresponding periods for analysis.

## 9 Conclusions and Future Work

Troubleshooting native cloud applications requires profound domain knowledge and essential skills due to the complexity of distributed systems with nonlinear probabilistic interrelations mostly hidden without modern APM solutions. The latest collects and stores all available information like time series, logs, traces, and events for detection, identification, and remediation of IT issues. Timely resolution of performance degradations is realistic only with ML/AI-empowered data analytics. However, system administrators require interpretable /explainable innovative solutions otherwise, no one will blindly follow black-box recommendations.

The paper's main goal is to nourish distributed tracing with explainable RCA for accelerating the resolution of performance degradations. Distributed tracing is a method of application monitoring based on microservices architecture. It provides better visibility of application components compared to classical monitoring based on time series and logs. The atoms of information are traces that describe the requests flowing through the different microservices of an application. They comprise spans and tags for more detailed monitoring of hidden processes. The analysis of a malfunctioning microservice can be performed via trace-traffic passing through that component.

We considered applying a classification rule-induction approach known as RIPPER to the trace traffic to reveal the explanations of performance degradations that can accelerate the mean time to resolution in such complex environments. The explanations can be derived from rules containing information regarding the spans, tags, and their values. We suggested three different algorithms that swept the trace traffic with different resolutions. Algorithm A worked with the lowest resolution, incorporating trace types and spans. Algorithm B utilized the tags and their values. Algorithm C combined both approaches. The latest internally selected the required level of resolution.

We suggested trace labeling based on different indicators. One of the approaches performed data labeling based on the internal tag known as "error." Its value differentiates typical and erroneous traces. Another approach utilized trace durations, assuming that similar trace types had almost identical durations, and violation of that principle should be the reason for the inspection. Finally, we considered data labeling based on specific tag values of spans that should be interesting to explain different tag categories, like "error.type."

We experimented with customer data and illustrated the benefits of different scenarios. We proved that the outcomes of RIPPER were highly explainable, and the execution times were acceptable, even for trace traffic with a large number of spans and tags. RIPPER has technical advantages like tolerance towards missing values, flexibility to work with multi-class problems, and the capability to consume numeric and categorical variables. This approach was a part of patented analytics designed for native cloud applications. It was productized by VMware and passed multi-layer validation steps.

The explainability and predictive power of ML algorithms have opposite directions. As a highly explainable approach, RIPPER has comparatively lower predictive power than neural networks, boosting, SVM, and others. In some situations, the classification accuracy will be insufficient for relying on the corresponding recommendations. We will investigate applying more powerful approaches to the problem, which can predict the appearance of erroneous traces with more extensive coverage and confidence. However, due to a lack of explainability, we must explore XAI possibilities (see [Harel et al., 2022, Hooker et al., 2018]) to enhance this disadvantage. Probable solutions can be the application of the SHAP method (see [Lundberg and Lee, 2017, Mayer, 2022, Shapley, 1953]), LIME (see [Ribeiro et al., 2016]), built-in feature importance analysis in the

boosting methods (see [Sandri and Zuccolotto, 2008]) and model agnostic permutations-based approach (see [Breiman, 2001, Altmann et al., 2010]). Another interesting approach is incorporating user feedback into the process of recommendation prioritization.

### Acknowledgements

The RA Science Committee funded Arnak Poghosyan in the frames of the research project № 20TTAT-AIa014. The research is conducted within the ADVANCE Research Grants provided by the Foundation for Armenian Science and Technology.

We thank anonymous reviewers for the critical reading of our paper. Their insightful comments helped to improve the manuscript.

### References

- [Agrawal et al., 1993] Agrawal, R., Imieliński, T., and Swami, A. (1993). Mining association rules between sets of items in large databases. In *Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, pages 207–216.
- [Altmann et al., 2010] Altmann, A., Toloşi, L., Sander, O., and Lengauer, T. (2010). Permutation importance: a corrected feature importance measure. *Bioinformatics*, 26(10):1340–1347.
- [Barredo Arrieta et al., 2020] Barredo Arrieta, A., Diaz-Rodriguez, N., Del Ser, J., Bennetot, A., Tabik, S., Barbado, A., Garcia, S., Gil-Lopez, S., Molina, D., Benjamins, R., Chatila, R., and Herrera, F. (2020). Explainable artificial intelligence (XAI): Concepts, taxonomies, opportunities and challenges toward responsible AI. *Information Fusion*, 58:82 – 115.
- [Breiman, 2001] Breiman, L. (2001). Random forests. *Mach. Learn.*, 45(1):5–32.
- [Cai et al., 2019] Cai, Z., Li, W., Zhu, W., Liu, L., and Yang, B. (2019). A real-time trace-level root-cause diagnosis system in Alibaba datacenters. *IEEE Access*, 7:142692–142702.
- [Cohen, 1995] Cohen, W. W. (1995). Fast effective rule induction. In *Proceedings of the Twelfth International Conference on Machine Learning*, pages 115–123. Morgan Kaufmann.
- [Dunning and Ertl, 2019] Dunning, T. and Ertl, O. (2019). Computing extremely accurate quantiles using t-digests.
- [Elsaadawy et al., 2019] Elsaadawy, M., Kemme, B., and Younis, M. (2019). Enabling efficient application monitoring in cloud data centers using sdn.
- [Fürnkranz, 1997] Fürnkranz, J. (1997). Pruning algorithms for rule learning. *Mach. Learn.*, 27(2):139–172.
- [Fürnkranz et al., 2012] Fürnkranz, J., Gamberger, D., and Lavrač, N. (2012). *Foundations of rule learning*. Cognitive Technologies. Springer, Heidelberg. With a foreword by Geoffrey I. Webb.
- [Fürnkranz and Kliegr, 2015] Fürnkranz, J. and Kliegr, T. (2015). A brief overview of rule learning. In Bassiliades, N., Gottlob, G., Sadri, F., Paschke, A., and Roman, D., editors, *Rule technologies: Foundations, tools, and applications*, pages 54–69, Cham. Springer International Publishing.
- [Fürnkranz and Widmer, 1994] Fürnkranz, J. and Widmer, G. (1994). Incremental reduced error pruning. In *Proc. 11th International Conference on Machine Learning*, pages 70–77. Morgan Kaufmann.
- [Han et al., 2004] Han, J., Pei, J., and Yin, Y. (2004). Mining frequent patterns without candidate generation: A frequent-pattern tree approach. *Data Mining and Knowledge Discovery*, 8:53–87.

- [Harel et al., 2022] Harel, N., Gilad-Bachrach, R., and Obolski, U. (2022). Inherent inconsistencies of feature importance.
- [Harutyunyan et al., 2022] Harutyunyan, A., Aghajanyan, N., Harutyunyan, L., Poghosyan, A., Bunarjyan, T., and Han Vinck, A. (2022). On diagnosing cloud applications with explainable ai. In Hajian, A., Baloian, N., Inoue, T., and Luther, W., editors, *Third CODASSCA Workshop, Yerevan, Armenia: Collaborative Technologies and Data Science in Artificial Intelligence Applications*, pages 23–26, Berlin. Logos Verlag.
- [Harutyunyan et al., 2020a] Harutyunyan, A. N., Grigoryan, N. M., and Poghosyan, A. V. (2020a). Fingerprinting data center problems with association rules. In Hajian, A., Baloian, N., Inoue, T., and Luther, W., editors, *Proceedings of the Second CODASSCA Workshop, Yerevan, Armenia: Collaborative Technologies and Data Science in Artificial Intelligence Applications*, pages 152–158, Berlin. Logos Verlag.
- [Harutyunyan et al., 2020b] Harutyunyan, A. N., Grigoryan, N. M., Poghosyan, A. V., Dua, S., Antonyan, H., Aghajanyan, K., and Zhang, B. (2020b). Intelligent troubleshooting in data centers with mining evidence of performance problems. In Hajian, A., Baloian, N., Inoue, T., and Luther, W., editors, *Proceedings of the Second CODASSCA Workshop, Yerevan, Armenia: Collaborative Technologies and Data Science in Artificial Intelligence Applications*, pages 169–180, Berlin. Logos Verlag.
- [Harutyunyan et al., 2019] Harutyunyan, A. N., Poghosyan, A. V., Grigoryan, N. M., Hovhannisyan, N. A., and Kushmerick, N. (2019). On machine learning approaches for automated log management. *J. Univers. Comput. Sci. (JUCS)*, 25(8):925–945.
- [Harutyunyan et al., 2018] Harutyunyan, A. N., Poghosyan, A. V., Kushmerick, N., and Grigoryan, N. (2018). Learning baseline models of log sources. In Hajian, A., Luther, W., and Vinck, A. J. H., editors, *Proceedings of the CODASSCA Workshop, Yerevan, Armenia: Collaborative Technologies and Data Science in Artificial Intelligence Applications*, pages 145–156, Berlin. Logos Verlag.
- [Heger et al., 2017] Heger, C., van Hoorn, A., Mann, M., and Okanović, D. (2017). Application performance management: State of the art and challenges for the future. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering, ICPE '17*, page 429–432, New York, NY, USA. Association for Computing Machinery.
- [Helmer et al., 1998] Helmer, G., Wong, J., Honavar, V., and Miller, L. (1998). Intelligent agents for intrusion detection. In *Proceedings IEEE Information Technology Conference, Syracuse, NY*, pages 121–124. Springer.
- [Helmer et al., 2002] Helmer, G., Wong, J. S., Honavar, V., and Miller, L. (2002). Automated discovery of concise predictive rules for intrusion detection. *Journal of Systems and Software*, 60(3):165–175.
- [Hooker et al., 2018] Hooker, S., Erhan, D., Jan Kindermans, P., and Kim, B. (2018). Evaluating feature importance estimates. *arXiv*.
- [Hühn and Hüllermeier, 2009] Hühn, J. and Hüllermeier, E. (2009). FURIA: An algorithm for unordered fuzzy rule induction. *Data Min. Knowl. Discov.*, 19(3):293–319.
- [John et al., 1994] John, G. H., Kohavi, R., and Pfleger, K. (1994). Irrelevant features and the subset selection problem. In *Machine Learning: Proceedings of the 11th International Conference*, pages 121–129. Morgan Kaufmann.
- [Klaise et al., 2020] Klaise, J., Van Looveren, A., Cox, C., Vacanti, G., and Coca, A. (2020). Monitoring and explainability of models in production.
- [Knoche and Hasselbring, 2019] Knoche, H. and Hasselbring, W. (2019). Drivers and barriers for microservice adoption – a survey among professionals in germany. *Enterprise Modelling and Information Systems Architectures (EMISAJ) – International Journal of Conceptual Modeling*, 14(1):1–35.

- [Lee and Stolfo, 1998] Lee, W. and Stolfo, S. J. (1998). Data mining approaches for intrusion detection. In *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7*, SSYM'98, page 6, USA. USENIX Association.
- [Lin et al., 2020] Lin, F., Muzumdar, K., Laptev, N. P., Curelea, M.-V., Lee, S., and Sankar, S. (2020). Fast dimensional analysis for root cause investigation in a large-scale service environment. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 4(2):1–23.
- [Lin et al., 2018] Lin, J., Chen, P., and Zheng, Z. (2018). Microscope: Pinpoint performance issues with causal graphs in micro-service environments. In *ICSOC*.
- [Liu et al., 2021] Liu, D., He, C., Peng, X., Lin, F., Zhang, C., Gong, S., Li, Z., Ou, J., and Wu, Z. (2021). Microhecl: High-efficient root cause localization in large-scale microservice systems. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 338–347.
- [Liu and Motoda, 1998] Liu, H. and Motoda, H. (1998). *Perspectives of Feature Selection*, pages 17–41. Springer US, Boston, MA.
- [Lundberg and Lee, 2017] Lundberg, S. and Lee, S.-I. (2017). A unified approach to interpreting model predictions.
- [Mahmud et al., 2021] Mahmud, R., Ramamohanarao, K., and Buyya, R. (2021). Application management in fog computing environments. *ACM Computing Surveys*, 53(4):1–43.
- [Mannila et al., 1995] Mannila, H., Toivonen, H., and Verkamo, A. I. (1995). Discovering frequent episodes in sequences extended abstract. In *Proceedings of the First International Conference on Knowledge Discovery and Data Mining*, KDD'95, page 210–215. AAAI Press.
- [Marvasti et al., 2013] Marvasti, M. A., Poghosyan, A. V., Harutyunyan, A. N., and Grigoryan, N. M. (2013). Pattern detection in unstructured data: An experience for a virtualized IT infrastructure. In Turck, F. D., Diao, Y., Hong, C. S., Medhi, D., and Sadre, R., editors, *2013 IFIP/IEEE International Symposium on Integrated Network Management, IM 2013, Ghent, Belgium, May 27-31, 2013*, pages 1048–1053. IEEE.
- [Mayer, 2022] Mayer, M. (2022). Shap for additively modeled features in a boosted trees model.
- [Nag et al., 2021] Nag, D. A., Grigoryan, N. M., Poghosyan, A. V., and Harutyunyan, A. N. (2021). Methods and systems that identify dimensions related to anomalies in system components of distributed computer systems using traces, metrics, and component-associated attribute values. Patent US US11113174. Filed March 27, 2020. Issued Sep 7, 2021.
- [Notaro et al., 2020] Notaro, P., Cardoso, J., and Gerndt, M. (2020). A systematic mapping study in aiops.
- [Parker et al., 2020] Parker, A., Spoonhower, D., Mace, J., Sigelman, B., and Isaacs, R. (2020). *Distributed Tracing in Practice: Instrumenting, Analyzing, and Debugging Microservices*. O'Reilly Media, Incorporated.
- [Poghosyan et al., 2021a] Poghosyan, A., Ashot, N., G.M., N., and Kushmerick, N. (2021a). Incident management for explainable and automated root cause analysis in cloud data centers. *JUCS - Journal of Universal Computer Science*, 27(11):1152–1173.
- [Poghosyan et al., 2021b] Poghosyan, A., Harutyunyan, A., Grigoryan, N., Pang, C., Oganessian, G., Ghazaryan, S., and Hovhannisyan, N. (2021b). An enterprise time series forecasting system for cloud applications using transfer learning. *Sensors*, 21(5).
- [Poghosyan et al., 2016] Poghosyan, A. V., Harutyunyan, A. N., and Grigoryan, N. M. (2016). Managing cloud infrastructures by a multi-layer data analytics. In Kounev, S., Giese, H., and Liu, J., editors, *2016 IEEE International Conference on Autonomic Computing, ICAC 2016, Wuerzburg, Germany, July 17-22, 2016*, pages 351–356. IEEE Computer Society.
- [Poghosyan et al., 2022] Poghosyan, A. V., Harutyunyan, A. N., Grigoryan, N. M., and Pang, C. (2022). Root cause analysis of application performance degradations via distributed tracing. In

- Hajian, A., Baloian, N., Inoue, T., and Luther, W., editors, *Proceedings of the CODASSCA Workshop, Yerevan, Armenia: Collaborative Technologies and Data Science in Artificial Intelligence Applications*, pages 27–31, Berlin. Logos Verlag.
- [Poghosyan et al., 2021c] Poghosyan, A. V., N., H. A., Grigoryan, N. M., Pang, C., Oganesyanyan, G., and Baghdasaryan, D. (2021c). Automated methods and systems that facilitate root cause analysis of distributed-application operational problems and failures. Patent US Application No.: 17/491,967 and 17/492,099. Filed Oct 1, 2021.
- [Quinlan, 2014] Quinlan, J. R. (2014). *C4.5: programs for machine learning*. Elsevier.
- [Ribeiro et al., 2016] Ribeiro, M. T., Singh, S., and Guestrin, C. (2016). "why should i trust you?": Explaining the predictions of any classifier.
- [Sandri and Zuccolotto, 2008] Sandri, M. and Zuccolotto, P. (2008). A bias correction algorithm for the gini variable importance measure in classification trees. *Journal of Computational and Graphical Statistics*, 17(3):611–628.
- [Shapley, 1953] Shapley, L. S. (1953). *17. A Value for n-Person Games*, pages 307–318. Princeton University Press, Princeton.
- [Solé et al., 2017] Solé, M., Muntés-Mulero, V., Rana, A. I., and Estrada, G. (2017). Survey on models and techniques for root-cause analysis. ArXiv:1701.08546.
- [Suriadi et al., 2013] Suriadi, S., Ouyang, C., van der Aalst, W., and ter Hofstede, A. (2013). Root cause analysis with enriched process logs. In Rosa, M. L. and Soffer, P., editors, *Business Process Management Workshops, International Workshop on Business Process Intelligence (BPI 2012). Volume 132 of Lecture Notes in Business Information Processing*, pages 174–186, Berlin. Springer-Verlag.
- [Tzanettis et al., 2022] Tzanettis, I., Androna, C.-M., Zafeiropoulos, A., Fotopoulou, E., and Papavassiliou, S. (2022). Data fusion of observability signals for assisting orchestration of distributed applications. *Sensors*, 22(5).
- [Vitali, 2022] Vitali, M. (2022). Towards greener applications: Enabling sustainable cloud native applications design.
- [VMware, 2022] VMware (2022). Distributed tracing overview. [https://docs.wavefront.com/tracing\\_basics.html](https://docs.wavefront.com/tracing_basics.html). Accessed: 2022-12-1.
- [Witten et al., 2005] Witten, I. H., Frank, E., Hall, M. A., and Pal, C. J. (2005). Practical machine learning tools and techniques. *Morgan Kaufmann*.