Root Cause and Liability Analysis in the Microservices Architecture for Edge IoT Services

Onur Kalinagac*, Wissem Soussi*, Yacine Anser‡, Chrystel Gaber§ and Gürkan Gür*
* Zurich University of Applied Sciences (ZHAW) InIT, Winterthur, Switzerland, name.surname@zhaw.ch
‡Orange & Cnam, Caen, France, yacine.anser@orange.com
§Orange, Chatillon, France, chrystel.gaber@orange.com

***Abstract*—In this work, we present a liability analysis framework for root cause analysis (RCA) in the microservices architecture with IoT-oriented containerized network services. We keep track of the performance metrics of microservices, such as service response time, memory usage and availability, to detect anomalies. By injecting faults in the services, we construct a Causal Bayesian Network (CBN) which represents the relation between service faults and metrics. Service Level Agreement (SLA) data obtained from a descriptor named TRAILS (sTakeholder Responsibility, AccountabIlity and Liability deScriptor) is also used to flag service providers which have failed their commitments. In the case of SLA violation, the constructed CBN is used to predict the fault probability of services under given metric readings and to identify the root cause.**

Keywords*—*Anomaly detection, root cause analysis, microservices, Service Level Agreement, Causal Bayesian Network.

I. Introduction

In recent years, the complexity of software systems has increased vastly, and monolithic systems are becoming harder to build and maintain. Microservices architecture allows a large application to be divided into smaller, independent parts, each with its own set of responsibilities. This paradigm has become an architectural pattern for service-based cloud computing environments as well as mobile networks such as Beyond 5G or 6G systems. For the latter, those granular services are envisaged to serve an immense number of users and IoT devices in the edge-to-cloud continuum. They can be implemented by different service providers using different technologies or programming languages as envisaged in multi-party, multi-stakeholder, and open future network environments. A microservices-based application can call on many internal microservices to compose its response in order to fulfill a single user request. Hence, in the event of a failure, the responsible party may be unclear, compromising liability and accountability [1].

For network and service management, it is a particular challenge to support confidence between parties and compliance with regulations for cloud and telecom infrastructures. In that regard, one of the proposed solutions is to use Service Level Agreements (SLAs) which are extensively investigated. They are contracts in which service providers guarantee the quality of the services by defining usage conditions [2]. SLAs provide a basic rule set; however, the cases where no violation occurs due to too-loose commitments or where multiple SLA violations occur due to internal dependencies may have an adverse effect on the accuracy of determining the true cause of a failure. To address this issue, we proposed a proof-of-concept demo application named Graph-Based Liability Analysis Framework (GRALAF) in [3]. It performs near real-time anomaly detection and root cause analysis (RCA) in a microservices environment.

In GRALAF, we construct a Causal Bayesian Network (CBN) with NOTEARS [4] algorithm from a dataset obtained by injecting faults to services. This dataset consists of the service metrics and service fault states. CBN helps us to learn about the causal relations between the fault status of services and their metrics. Then, we try to detect meaningful changes in the performance metrics of microservices or an SLA violation. When either of them happens, we predict the corresponding state of services in terms of being faulty. By sorting our findings, we conclude the root cause of the failure. These findings are sent to an external liability service for future investigation.

In this paper, we improve our graph-based liability analysis framework for microservices. We implement clustering based anomaly detection with Gaussian mixture models and introduce different fault types including performance, reliability, and traffic. We also evaluate our framework's performance and scalability under different conditions such as multiple service failures. The rest of the paper is organized as follows. First, we provide technical background information about CBN and an overview of related studies in anomaly detection and RCA. In Section III, we describe the system model and its components. Then, we present the performance metrics and evaluation results in Section IV. After discussing the challenges faced during this work in Section V, we conclude the paper in Section VI.

II. Technical Background and Related Work

A. Technical Background

Bayesian Networks: A Bayesian network (BN) is a probabilistic graphical model for representing joint probability distributions [5]. A directed acyclic graph (DAG) represents a set of variables and their conditional dependencies and conditional probability distributions (CPDs) over each variable, given that its parents are the two main components of a Bayesian network. As DAGs, Bayesian networks provide a useful way to represent causal relations. They depict the direct dependencies between variables, or in the case of a causal BN, the direct cause and effect relationships.

Structure Learning: One way of creating a Bayesian network is by learning it from a dataset; and this is also

DOI: 10.1109/ICC45041.2023.10279721

how we generate the BN in this study. In general, structure learning algorithms are divided into two categories [6]. The first type is constraint-based, which eliminates and orients edges based on a series of conditional independence (CI) tests. The second class, score-based methods, is a traditional machine learning approach in which the goal is to search over different graphs in order to maximize an objective function. The number of possible graphs for the given number of nodes increases exponentially. There are potential three DAGs for 2 nodes, compared to 7.8730×10^{11} for 8 nodes. Both of the approaches are proved to be NP-hard [7]. Moreover, when learning structures from data, the possibility of different graph structures representing the same independence relations is problematic. There are multiple structures that can accurately represent the independence relations found in the data in general, so expert knowledge is commonly used.

Inference: After constructing a BN, CPDs can be used to predict the state of a node as well as the probability of each possible state of a node by giving some input data. Even though calculations take some time, heuristics like the Junction Tree algorithm [8] make it faster each time a prediction request, namely *query*, is made by caching the intermediate calculations. Many intermediate factors are produced as a byproduct of the main computation; these factors turn out to be the same as those required to answer other queries.

B. Related Work

There are many research works dealing with anomaly detection and graph-based RCA in the literature [9]. Here, we particularly discuss the ones based on the microservices architecture.

Studies including a causality graph-based analysis mostly use PC algorithm [10] to build a causality graph. Microscope proposed in [11] uses standard deviation-based heuristics to detect anomalies in response times on front-end requests. In anomaly detection, they use the 3-sigma rule to identify outliers. Starting from the application front-end, their algorithm recursively visits the nodes of the causality graph in the opposite direction of edges. It looks at a node's neighbors if it exhibits abnormal behavior. In the last step, the root cause candidates are ranked according to their anomaly score or the Pearson correlation between their response times and the application front-end.

In [12], the MicroRCA algorithm uses topology graph-based analysis and creates a topology graph with vertices representing services and host machines, and oriented arcs representing service interactions and hosting. Each vertex of the topology graph is associated with the time series of KPIs monitored on the relevant service or node. The MicroRCA uses BIRCH (Balanced Iterative Reducing and Clustering using Hierarchies) algorithm to detect anomalies. The RCA is carried out by modifying the random walk-based search in the subgraph suggested in MonitorRank [13]. In [14], they perform causality graph-based RCA for the performance anomalies based on CPU and memory stress in the microservices environment. They utilize Sock-shop microservices application [15] as in MicroRCA.

Thalteim et al. [16] try to reduce the utilized KPI metrics to decrease the RCA overhead. Their application collects information on service interactions from network system calls, which is then used to automatically reconstruct the topology of a running application. When a performance anomaly is detected on the application front-end, the topology is used to drive the RCA. It begins by removing KPIs from consideration whose variance is too small to have any statistical significance and by grouping the KPIs tracked by each service so that only one representative KPI is taken into account for each cluster.

MicroHECL [17] constructs a topology graph by tracking interaction traces, which include the beginning and ending times of service interactions, their source and target services, performance indicators, and the particular user request's unique identifier. It determines the anomaly propagation chains that could have led to the observed anomaly based on this topology. The possible root causes are ranked based on the Pearson correlation between their performance metrics and those of the service where the performance anomaly was first observed. They group anomalies under performance, reliability, and traffic categories.

III. System Model

As our use case, we choose an IoT device management scenario in which IoT sensors are connected to edge services and their data are sent to a remote server, as illustrated in Fig. 1. This edge service consists of several microservices while GRALAF tracks anomalies and performs the RCA for these services in case of an anomaly detection.

A. Architecture Description

Our system diagram is shown in Fig. 2. In this block diagram, you can see how the system components interact with each other. Their operations and interconnections are further detailed as follows.

GRALAF: GRALAF tracks metrics and compares them with the given SLA data from TRAILS. In the case of an SLA violation, it performs RCA based on CBN and reports to an external liability service about the corresponding violation with the estimated probability of fault types for each service being responsible for the incident. It is developed in Python and deployed in the same Kubernetes environment with the Edgex services. The detailed workflow is explained in the following subsection.

TRAILS: TRAILS (sTakeholder Responsibility, AccountabIity and Liability deScriptor) [26] is a metamodel that extends the TOSCA NFV metamodel [27] by merging the existing profile in the cloud-to-IoT continuum such as the MUD (Manufacturer Usage Definition) profile [28] and the VNF (Virtual Network Function) descriptor [29], and adding a description of the responsibilities, accountabilities and liabilities of supply chain actors.

With TRAILS, several actors involved in the creation of a service can define their commitments independently of each other. Service providers can describe the promised functionality of a service; for example; an IoT manufacturer who proposes a device that provides Bluetooth connectivity or a service provider that promises a service with an availability of 99% can commit to it. Service providers can also define usage conditions under which commitments are valid; for example,

DOI: 10.1109/ICC45041.2023.10279721

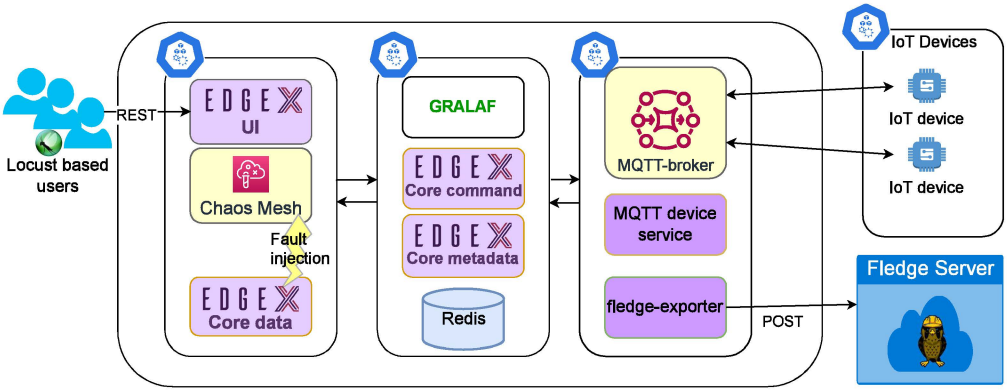


Fig. 1. IoT management use case.

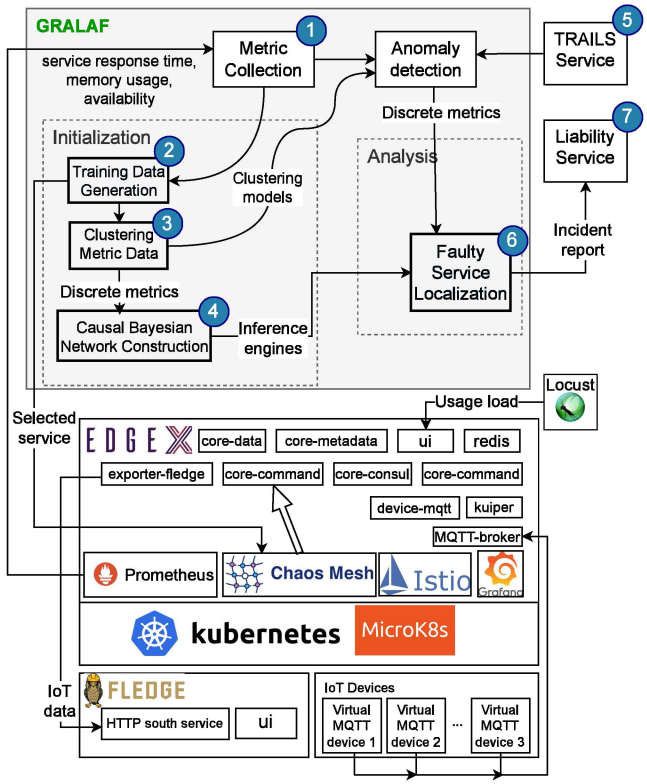


Fig. 2. System block diagram for the test environment.

a VNF provider can add usage conditions to specify the hardware configuration (CPU core and memory capacity) for the VNF to behave as expected. A validator can indicate when a component/service has been validated and the result of the validation. TRAILS distinguishes between authors who take responsibility for a specific property and authors (lead author) who integrate several services provided by other authors. The lead author can define the inter-dependencies of the services that compose the global service using a data structure defined in the metamodel. TRAILS is generic, which means that it can be used to describe an IoT device, a VNF, or a microservice. The commitments are formalized in TRAILS through the SLA which is a contract that defines an agreement between two parties: the user and the service provider. It describes the terms and conditions of the service delivery. On the user side, it identifies the requirements of the user. On the provider side, it indicates the commitments (based on capabilities) of the provider toward the customer [30].

TABLE I. SYSTEM COMPONENTS.

Component	Description and Role
Kubernetes	Kubernetes MicroK8s distribution [18] is used to deploy the following microservices.
Virtual MQTT device	We developed a dummy IoT device application which can be controlled through MQTT. It also periodically sends random sensor data. It is coded in Python and dockerized for easy scalability.
Edgex	Edgex [19] is an open source software framework that provides interoperability between devices and application at the IoT edge. We deployed its main services on our Kubernetes environment. MQTT-broker indirectly connects MQTT based device service with the virtual MQTT devices. A data exporter app named `exporter-fledge` is also deployed to send sensor data to Fledge server.
Fledge	Fledge [20] is an open source framework and community for the industrial edge focused on IoT devices. It is mainly used for fog computing. In our system, it operates as cloud server application where the data exported from Edgex.
Istio	As a service mesh, it allows transparently adding capabilities like observability, traffic management, and security without adding them to service code [21].
Prometheus	As a time series database, it periodically scrapes system related data from Kubernetes pods, node exporter and Istio service [22].
Chaos Mesh	We use Chaos Mesh [23] to create delay, memory and availability fault injections to our microservices through its REST API for Kubernetes.
Locust	Locust [24] is an open source performance testing tool that can simulate a large number of concurrent users. We developed a Python script which uses its library to generate multi-threaded load on the REST endpoints of the Edgex UI.
Kiali	An Istio observability console that supports service mesh configuration and validation. Used to demonstrate the effect of fault injections on services' metrics [25].

The remaining components of the system are listed and described in Table I. All the GRALAF related applications can be found in our GitHub repository https://github.com/INSPIRE-5Gplus/gralaf.

B. Description of the GRALAF Workflow

The GRALAF building blocks and operation phases for RCA are shown in the upper part of Fig. 2.

(1) In order to monitor the operation of services, GRALAF periodically queries Prometheus for microservice metrics including the response time, availability, and memory. We start creating our training dataset with the initial 30 readings. We also add another attribute for the fault status of the services. Services are assumed to represent their normal working conditions at this initial phase, so their fault statuses are set to 0.

(2) In the second phase, we start injecting delay, availability, and memory faults into a microservice in a pre-ordered way through Chaos Mesh and we save the corresponding

DOI: 10.1109/ICC45041.2023.10279721

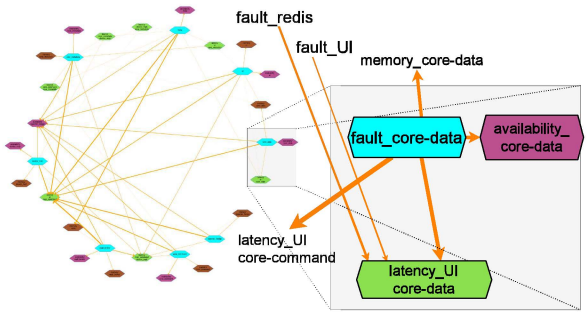


Fig. 3. Constructed CBN after GRALAF training.

values. The fault status of the selected microservice is set to $x \in \{1, 2, 3\}$ depending on the fault type while the fault statuses of the microservices with no injection are set to 0 on each reading. This phase continues until we get enough data depending on the number of microservices.

3 In the third phase, we start with the normalization of all the metrics data except those already normalized, such as availability. Then, we discretize our metrics data by applying the Bayesian Gaussian Mixture model based on the Expectation–maximization algorithm [31]. Each metric is clustered separately with a given max cluster size $n_{cluster}$ and the raw metric readings are switched with the corresponding cluster IDs. In order to decrease the unbeneficial complexity, we remove some of the metrics according to the explanation given in Section III-C.

4 We train a structure model with the `CausalNex` Python package, which uses the NOTEARS algorithm. From that model, we construct the CBN by using the same training dataset. To increase prediction speed, we remove weak links from the graph. In Fig. 3, you can see the visualization of the CBN graph, with the edge thickness indicating the causal probability between the nodes. Nodes with a turquoise color indicate a service's fault state, while nodes with brown, purple or green coloring indicate a service's memory usage, its availability or response time between service pairs. This graph indicates that if an abnormal response time between UI and core-data services is observed, a core-data fault is much more likely to be the root cause than a UI service fault. For each independent CBN subgraph, an inference engine is constructed.

5 GRALAF retrieves the whole TRAILS data from an external TRAILS service. This step is triggered periodically so the latest version of the data is used.

These five phases complete our preparation part.

6 During the operation, when an SLA violation is detected in the metric readings, it is called an *incident*. When that occurs, GRALAF estimates the probability of the fault statuses of each service by using the previously constructed inference engines of the existing CBN.

7 For each service, the fault score is estimated by subtracting the estimated probability of the fault '0' from 1. If the top probability value is higher than a threshold value $\Theta_{decision}$, we accept it as an *incident*. In the case of an *incident*, the non-zero scores are ordered and the ones with probability value higher than $\Theta_{decision}$ are sent with the supplementary info such as discretized metric data and related date and time to an external liability service for further investigations. A sample report data is given in Listing.1.

Listing 1. Sample report data


```
{"violation_details": {"violation_time": "2022-10-31 14:39:24",
 "violation_type": "availability",
 "expected_value": {"min": 0.99, "max": 1.0}, "reported_value": 0.416,
 "contract_info": {"service": "edgex-ui","responsible_provider": "SP4"}},
 "root_causes": [
    {"service_name": "edgex_ui", "probability": 1.0,
      "fault_distribution": {"0": 0.0, "1":0.0, "2":0.0, "3":1.0}},
    {"service_name": "edgex_core_data", "probability": 0.0952,
      "fault_distribution": {"0": 0.905, "1": 0.048, "2":0.048, "3":0.0}},],
 "violation_evidence": {
    "latency_edgex_ui_edgex_core_metadata": 0,
    "latency_edgex_core_command_edgex_device_mqtt": 1,
    "latency_edgex_core_command_edgex_core_metadata": 0,
    "latency_edgex_ui_edgex_core_command": 1,
    "latency_edgex_device_mqtt_edgex_core_metadata": 0,
    "latency_edgex_ui_edgex_core_data": 0,
    "latency_edgex_core_metadata_edgex_device_mqtt": 0,
    "availability_edgex_exporter_fledge": 0,
    "memory_edgex_exporter_fledge": 0,
    "availability_edgex_redis": 0,
    "memory_edgex_redis": 1,
    "availability_edgex_core_data": 0,
    "memory_edgex_core_data": 0,
    "availability_edgex_ui": 1,
    "memory_edgex_ui": 0,
    "availability_edgex_device_mqtt": 0,
    "memory_edgex_device_mqtt": 0,
    "availability_edgex_core_metadata": 0,
    "memory_edgex_core_metadata": 0,
    "availability_edgex_mqtt_broker": 0,
    "memory_edgex_mqtt_broker": 0,
    "availability_edgex_core_command": 0,
    "memory_edgex_core_command": 0},
 "root_cause_analysis_time": "2022-10-31 14:39:59"
}
```


C. Metric Filtering

In order to decrease the complexity of the CBN generation, we discard some metrics depending on the following criteria. The first criterion is the availability of data where some metrics can not be retrieved at each step regularly. Because some of the service pairs do not interact regularly, the related response time is not calculated for all the intervals. By setting a threshold value $\Theta_{availability}$, we filter out these metrics with low availability. Secondly, some metrics do not respond to fault injections consequentially, so we consider them as insignificant. After clustering all the metrics data one by one, the ratio of the number of elements in the major cluster to all elements for the given metric is compared to a threshold value $\Theta_{insignificance}$. The metrics with a ratio higher than $\Theta_{insignificance}$ are discarded. The last criterion is reliability. We know the initial readings do not contain any fault injection, so we consider metric readings belonging to this part as ground-truth data and expect them to be clustered in the same group. If the ratio of readings clustered to the major cluster group in the initial readings is less than a threshold value $\Theta_{reliability}$, we consider the related metric unreliable and discard it from the dataset.

IV. Performance Evaluation

We utilize five VMs for the entire test setup in an OpenStack cloud infrastructure. Three of the VMs are used to deploy the MicroK8s environment which hosts Edgex and GRALAF microservices along with all the necessary system components such as Prometheus, Chaos Mesh, and Istio. Each of these three VMs has 4 vCPU, 8GB RAM, and 160GB SSD storage. One VM hosts a Fledge server while the other one hosts another MicroK8s environment where 25 MQTT-based virtual IoT device applications are deployed. The resource specifications for each of these two VMs are 1 vCPU, 2GB RAM, and 120GB SSD.

DOI: 10.1109/ICC45041.2023.10279721

TABLE II. TEST PARAMETERS.

Parameter	Value	Description
$n_{cluster}$	3	Max number of discrete values a metric can have based on clustering
$\Theta_{availability}$	0.7	Threshold value for filtering metrics with availability less than that
$\Theta_{insignificance}$	0.98	Threshold for filtering metrics with major discrete value frequency higher than that value
$\Theta_{reliability}$	0.7	Threshold for filtering metrics with the first cluster group ratio in the ground-truth data lower than that value
$\Theta_{decision}$	0.094-0.249	Threshold value for incident detection and if a service is responsible for the incident

We consider two different scenarios to evaluate our framework. In the first scenario, we analyze the cases in which there is only one faulty service at a time. It is the generic case used to evaluate RCA performance in many studies [11, 12, 14, 17]. In the second scenario, we investigate the simultaneous failures of different services to reflect more complex cases where the common resources are affected.

We have selected eight highly interdependent microservices from Edgex for RCA. We had two runs to create 50 separate datasets. Each has initialization readings which contain 30 steps without any fault injection and 72 steps with delay, memory, and pod failure injections (3×8 services $\times$ 3 fault types) ordered by *(service, fault type)* pair. For each injection, load levels are uniformly randomized. For the first group of the dataset, only one service fault is injected per reading step while two service faults are injected simultaneously to evaluate more complex RCA situations for the second one. After injecting a 70 seconds long fault, the system waits for 60 seconds before reading the corresponding metric. There is also a cool-down period of 15 seconds between each step, including the initial 30 reading steps. We chose these time values to see the full impact of a fault on system metrics since minimum query interval is one minute in Prometheus. One by one, we train the model with a single dataset from the first group. For the training data, the next dataset from the first group is selected for the first scenario, while datasets from the second group are used for the second scenario. From the test dataset, we choose 24 reading steps with no fault injection (N) and 24 steps with service failure (P).

In Table II, the threshold parameter values are listed. $\Theta_{decision}$ is estimated to maximize the accuracy for fault service detection by taking into consideration the optimum result of previous 10 datasets .

Accuracy, recall, and false positive ratio (FPR) are calculated for the detection of the correct faulty service. For the calculations, we count the incidents where the actual faulty services are ranked at the top of the RCA ranking as *true positive* (TP). The cases where no incident report is generated when there is no fault injection are counted as *true negative* (TN) and when there is actually an injection are counted as *false negative* (FN). For the accurate cases, we also check the success rate of the prediction of the fault type which is called as 'Fault Type Recall'. These metrics are calculated in the following way:

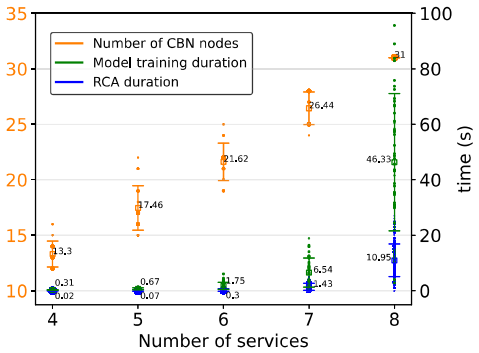


Fig. 4. Scalability based on the number of services considered for RCA. The number of CBN nodes is plotted against the left y-axis while the duration related metrics are plotted against the right y-axis.

$$Accuracy = (TP + TN)/(T + N) \quad (1)$$

$$Recall = TP/T \quad (2)$$

$$FPR = FP/N \quad (3)$$

Additionally, we use the top-k hit ratio (HR@k) and mean reciprocal rank (MRR) from MicroHECL [17]. We slightly modified them to adapt multi-service failure cases as well. HR@k denotes the probability that the actual root causes are found in the top-k result list. MRR is the average multiplicative inverse of the summation of rankings of the actual m faulty services in the prediction table. If any of these services is not available in the list, the rank sum can be regarded as positive infinity. The metrics can be represented as follows:

$$HR@k = \frac{1}{|A|} \sum_{i=1}^{A} (r_{ij} \in Rank_{ij}^{k}, \forall j) \quad (4)$$

$$MRR = \frac{1}{|A|} \sum_{i=1}^{A} \frac{j}{\sum_{1}^{m}(Index_{ij}^{k}) - j(j-1)/2} \quad (5)$$

where A is the number of incidents, r_{ij} is the jth root cause of the ith incident, $Rank_{ij}^{k}$ is the top k result list of the ith incident and $Index_{ij}$ is the rank of the jth root cause in the prediction list. Lastly, we investigate the effect of failure type on the RCA performance. Service problems are categorized into three anomaly types:

- Traffic Anomaly: delay injections on the service requests.
- Performance Anomaly: abnormal memory consumption injections on the service.
- Reliability Anomaly: pod failure injections.

A. Experimental Results

The average training time is measured as 46.33 seconds with a standard deviation of 25.61 while the average RCA analysis time was 10.95 seconds with a standard deviation of 4.64. In Fig. 4, you can see how these metrics change for the given number of services from 4 to 8. Estimated values of 50 datasets and its 2400 incidents are scattered for each different number of services. The average values are pinned in the graph while the standard deviation levels are also indicated with vertical lines on both sides of the averages. The number of

DOI: 10.1109/ICC45041.2023.10279721

services directly affects the number of nodes used to construct a CBN. As the number of nodes increases, the model training duration consisting of the construction of CBN and inference engine increases exponentially along with the RCA duration, which mainly refers to the utilization of the inference engine.

TABLE III. RESULTS FOR SINGLE SERVICE FAILURE.

Service	Accuracy	Recall	FPR	Fault type recall
All	0.943	0.919	0.033	0.698
core-command	0.980	0.987	0.027	0.642
ui	0.957	0.940	0.027	0.631
core-metadata	0.970	0.947	0.007	0.518
device-mqtt	0.977	0.987	0.033	0.687
mqtt-broker	0.983	0.987	0.020	**0.892**
exporter-fledge	**0.827**	**0.687**	0.033	0.796
core-data	0.963	0.973	0.047	0.637
redis	0.907	0.847	0.033	**0.825**

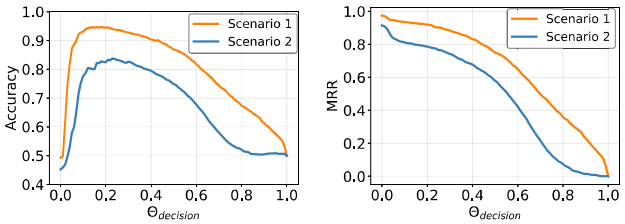
In Table III, we present the common classification metrics for the first scenario. The cause of the relatively lower accuracy for `exporter fledge` is that it is a client application where we can not track traffic-related faults quite well. Considering the low recall value, it can be said that we are not able to flag some problems with this kind of internal service. The challenge with fault type detection is that availability and traffic anomalies have some common effects on the system, making perfect detection difficult. On the other hand, we have higher fault type detection rates for infrastructure services such as `redis` and `mqtt-broker` because availability and delay anomalies cause much more distinct effects on the other services that have a client role for them.

TABLE IV. RESULTS FOR DOUBLE SERVICE FAILURES.

Service	Accuracy	Recall	FPR	Fault type recall
All	**0.799**	0.808	0.212	0.679
core-command	0.808	0.829	0.212	0.583
ui	0.829	0.855	0.197	0.599
core-metadata	0.806	0.831	0.220	0.534
device-mqtt	0.856	0.885	0.174	0.670
mqtt-broker	0.778	0.779	0.224	0.825
exporter-fledge	0.708	0.631	0.216	0.889
core-data	0.814	0.897	0.269	0.764
redis	0.774	0.752	0.203	0.628

We present the same metric calculations done for the second scenario in Table IV. Service rows represent the cases in which one of the failed services is the given one. We expect our application to find both of the actual failed services as top candidates. In terms of double service failure awareness, only in the 3.2% of incidents does GRALAF output either single service prediction or no incident decision in the case of multiple service failures. Since we try to adjust threshold value $\Theta_{decision}$ to cover all the responsible services, the threshold gets lower than the first scenario, which causes slightly high FPR, i.e., false alarms. On the other hand, we achieve nearly 80% of overall accuracy for double failures. In Fig. 5(a), accuracy estimations are given for different $\Theta_{decision}$ values. When $\Theta_{decision}$ is too low, the application starts to flag services as faulty even in no incident cases. Conversely, if it is too high, it rejects more of the service failure predictions. In Fig. 5(b), we can see that the same decision threshold filtering reduces the prediction list size, causing MRR to steadily decline.

In Table V, we can see the probable ranking value of the actual services.According to HR@1 value, the actual faulty service is our top suspect with a 92% probability. HR@3+



(a) Accuracy versus $\Theta_{decision}$. (b) MRR versus $\Theta_{decision}$.

Fig. 5. The effect of $\Theta_{decision}$ on the overall accuracy and MRR.

metrics are equal to HR@2 since GRALAF does not get many candidates for a given single service anomaly so they are skipped. As opposed to that, we see more potential incident candidates for the second scenario in Table VI because multiple failures cause much more unexpected outcomes on the metrics. HR@1 is zero because a single ranking does not cover double service failure at all.

Lastly, we present the same metrics based on the different anomaly types in Table VII and Table VIII. Traffic anomaly is relatively hard to detect since it may create cascaded metric anomalies while the correlated metrics for the other two anomaly types can be separated per service more easily.

TABLE V. THE HIT RATE RESULTS FOR SINGLE SERVICE FAILURE.

Service	HR@1	HR@2	MRR
All	**0.920**	0.933	0.925
core-command	0.987	0.987	0.986
ui	0.940	0.947	0.942
core-metadata	0.953	0.967	0.959
device-mqtt	0.980	0.987	0.983
mqtt-broker	0.993	0.993	0.998
exporter-fledge	0.693	0.693	0.685
core-data	0.987	0.987	0.986
redis	0.827	0.907	0.863

TABLE VI. THE HIT RATE RESULTS FOR DOUBLE SERVICE FAILURES.

Service	HR@2	HR@3	HR@4	MRR
All	0.748	0.808	0.817	0.789

TABLE VII. THE EVALUATION RESULTS BY ANOMALY TYPES.

Fault Type	Accuracy	Recall	FPR	Fault type recall
Traffic Anomaly	0.899	0.843	0.045	0.691
Performance Anomaly	0.959	0.963	0.045	0.629
Reliability Anomaly	0.981	0.995	0.033	0.726

V. DISCUSSION

The performance of anomaly detection has a massive impact on the RCA because it affects both the quality of CBN generation and fault service localization phases. As we experienced in [3], using metric mean value based thresholds for anomaly detection needs extra supervising since the variations of metrics are different. The standard deviation could be also used as an extra parameter to calculate better threshold values, but deciding on the number of states for each metric still cannot be achieved. To overcome this issue, we try to adapt unsupervised learning based clustering algorithms. In MicroRCA, they use BIRCH and it is quite powerful; however, it needs fine-tuning through its threshold parameter because the range of each metric can be very different, and the normalization process is really problematic due to the existence of outlier values. While selecting our clustering algorithm, we observed

DOI: 10.1109/ICC45041.2023.10279721

TABLE VIII. THE HIT RATE RESULTS BY ANOMALY TYPES.

Fault Type	HR@1	HR@2
Traffic Anomaly	0.835	0.843
Performance Anomaly	0.932	0.963
Reliability Anomaly	0.992	0.995

that the Gaussian mixture model based clustering algorithm outperformed BIRCH in our experiments since it handles outliers much better. Even though the clustering algorithm is inherently allowed to have more, it does not create many clusters, helping with the complexity issue.

Structure learning from data has significant scaling problems, as we mentioned in Section II-A. As the number of nodes increases over 35, learning time becomes unaffordable. Also, cycles in the graph start to emerge, which is against the DAG property of BNs. For 31 nodes, the constructed structure models get over 650 edges. Some of these edges can be removed after filtering them with heuristic algorithms. We forbid edges between metrics to drastically decrease complexity so that the final graph displays edges only between the service statuses and their correlated metrics.

VI. CONCLUSION

We proposed a graph liability analysis framework where we perform anomaly detection and CBN based RCA in a microservices environment. Also, we deployed an IoT edge network environment to develop and test our application. Our results indicate that RCA based on causality is very promising to get accurate analyses in small-to-mid scale infrastructures, while it suffers in the autonomous network generation phase as the number of services and variety of metrics increase. For future work, we will focus on the scalability of the framework as well as the enrichment of the reported RCA data.

ACKNOWLEDGMENT

The research leading to these results partly received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement no 871808 (5G PPP project INSPIRE-5Gplus). The paper reflects only the authors' views. The Commission is not responsible for any use that may be made of the information it contains.

REFERENCES

[1] C. Gaber, J. S. Vilchez, G. Gür, M. Chopin, N. Perrot, J.-L. Grimault, and J.-P. Wary, "Liability-aware security management for 5G," in *2020 IEEE 3rd 5G World Forum (5GWF)*, 2020, pp. 133–138.

[2] C.-Y. Lee, K. M. Kavi, R. A. Paul, and M. Gomathisankaran, "Ontology of secure service level agreement," in *2015 IEEE 16th International Symposium on High Assurance Systems Engineering*, 2015, pp. 166–172.

[3] O. Kalinagac, W. Soussi, and G. Gür, "Graph based liability analysis for the microservice architecture," in *2022 18th International Conference on Network and Service Management (CNSM)*, 2022, pp. 364–366.

[4] X. Zheng, B. Aragam, P. Ravikumar, and E. P. Xing, "Dags with no tears: Continuous optimization for structure learning," 2018. [Online]. Available: https://arxiv.org/abs/1803.01422

[5] M. de Jongh and M. J. Druzdzel, "A comparison of structural distance measures for causal bayesian network models," *Recent Advances in Intelligent Information Systems, Challenging Problems of Science, Computer Science series*, pp. 443–456, 2009.

[6] N. K. Kitson, A. C. Constantinou, Z. Guo, Y. Liu, and K. Chobtham, "A survey of bayesian network structure learning," *CoRR*, vol. abs/2109.11415, 2021. [Online]. Available: https://arxiv.org/abs/2109.11415

[7] D. M. Chickering, D. Heckerman, and C. Meek, "Large-sample learning of bayesian networks is NP-Hard," *J. Mach. Learn. Res.*, vol. 5, p. 1287–1330, Dec. 2004.

[8] A. L. Madsen and F. V. Jensen, "Lazy propagation: A junction tree inference algorithm based on lazy evaluation," *Artificial Intelligence*, vol. 113, no. 1, pp. 203–245, 1999.

[9] J. Soldani and A. Brogi, "Anomaly detection and failure root cause analysis in (micro) service-based cloud applications: A survey," *ACM Comput. Surv.*, vol. 55, no. 3, Feb 2022.

[10] P. Spirtes, C. N. Glymour, R. Scheines, and D. Heckerman, *Causation, prediction, and search*. MIT press, 2000.

[11] J. Lin, P. Chen, and Z. Zheng, "Microscope: Pinpoint performance issues with causal graphs in micro-service environments," in *Service-Oriented Computing*, C. Pahl, M. Vukovic, J. Yin, and Q. Yu, Eds. Cham: Springer International Publishing, 2018, pp. 3–20.

[12] L. Wu, J. Tordsson, E. Elmroth, and O. Kao, "MicroRCA: Root cause localization of performance issues in microservices," in *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*. IEEE, Apr. 2020, pp. 1–9.

[13] M. Kim, R. Sumbaly, and S. Shah, "Root cause detection in a service-oriented architecture," *ACM SIGMETRICS Performance Evaluation Review*, vol. 41, no. 1, pp. 93–104, 2013.

[14] L. Wu, J. Tordsson, J. Bogatinovski, E. Elmroth, and O. Kao, "Microdiag: Fine-grained performance diagnosis for microservice systems," in *2021 IEEE/ACM International Workshop on Cloud Intelligence (CloudIntelligence)*, 2021, pp. 31–36.

[15] Weaveworks, Container Solutions, "Sock shop: A microservices demo application," https://microservices-demo.github.io/.

[16] J. Thalheim, A. Rodrigues, I. E. Akkus, P. Bhatotia, R. Chen, B. Viswanath, L. Jiao, and C. Fetzer, "Sieve: Actionable insights from monitored metrics in distributed systems," in *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, 2017, p. 14–27.

[17] D. Liu, C. He, X. Peng, F. Lin, C. Zhang, S. Gong, Z. Li, J. Ou, and Z. Wu, "MicroHECL: High-efficient root cause localization in large-scale microservice systems," in *Proceedings of the 43rd International Conference on Software Engineering: Software Engineering in Practice*, 2021, p. 338–347.

[18] "MicroK8s homepage," https://microk8s.io, 2022.

[19] "EdgeX Foundry homepage," https://www.edgexfoundry.org, 2022.

[20] "Fledge project page," https://www.lfedge.org/projects/fledge, 2022.

[21] "Istio homepage," https://istio.io, 2022.

[22] "Prometheus homepage," https://prometheus.io, 2022.

[23] "Chaos Mesh homepage," https://www.chaos-mesh.org, 2022.

[24] "Locust homepage," https://locust.io, 2022.

[25] "Kiali homepage," https://kiali.io/, 2022.

[26] Y. Anser, C. Gaber, J.-P. Wary, S. N. M. García, and S. Bouzefrane, "TRAILS: Extending TOSCA NFV profiles for liability management in the Cloud-to-IoT continuum," in *2022 IEEE 8th International Conference on Network Softwarization (NetSoft)*, 2022, pp. 321–329.

[27] TOSCA simple profile for network functions virtualization (NFV)—version 1.0. [Online]. Available: http://docs.oasis-open.org/tosca/tosca-nfv/v1.0/tosca-nfv-v1.0.html

[28] E. Lear, R. Droms, and D. Romascanu, "Manufacturer Usage Description Specification," RFC 8520, Mar. 2019. [Online]. Available: https://www.rfc-editor.org/info/rfc8520

[29] ETSI, "ETSI GS NFV-IFA 011: Network functions virtualisation (NFV) management and orchestration VNF descriptor and packaging specification," https://www.etsi.org/deliver/etsi_gs/NFV-IFA/001_099/011/04.02.01_60/gs_NFV-IFA011v040201p.pdf, Group Specification ETSI GS NFV-IFA 011 v4.2.1, May 2021.

[30] ——, "ETSI EG 202 009-3 : User group; quality of ict services part 3: Template for service level agreements (sla)," https://www.etsi.org/deliver/etsi_eg/202000_202099/20200903/01.03.01_60/eg_20200903v010301p.pdf, Group Specification ETSI EG 202 009-3, July 2015.

[31] T. Moon, "The expectation-maximization algorithm," *IEEE Signal Processing Magazine*, vol. 13, no. 6, pp. 47–60, 1996.

DOI: 10.1109/ICC45041.2023.10279721