Prism: Revealing Hidden Functional Clusters from Massive Instances in Cloud Systems

Jinyang Liu[†*], Zhihan Jiang[†*], Jiazhen Gu[†], Junjie Huang[†], Zhuangbin Chen[‡**],
Cong Feng[§], Zengyin Yang[§], Yongqiang Yang[§], Michael R. Lyu[†]

[†]The Chinese University of Hong Kong, Hong Kong SAR, China, {jyliu, zhjiang22, jzgu, jjhuang23, lyu}@cse.cuhk.edu.hk
[‡]School of Software Engineering, Sun Yat-sen University, Zhuhai, China, chenzhb36@mail.sysu.edu.cn
[§]Computing and Networking Innovation Lab, Huawei Cloud Computing Technology Co., Ltd, China,
{fengcong5, yangzengyin, yangyongqiang}@huawei.com

Abstract—**Ensuring the reliability of cloud systems is critical for both cloud vendors and customers. Cloud systems often rely on virtualization techniques to create instances of hardware resources, such as virtual machines. However, virtualization hinders the observability of cloud systems, making it challenging to diagnose platform-level issues. To improve system observability, we propose to infer *functional clusters* of instances, *i.e.*, groups of instances having similar functionalities. We first conduct a pilot study on a large-scale cloud system, *i.e.*, Huawei Cloud, demonstrating that instances having similar functionalities share similar *communication* and *resource usage* patterns. Motivated by these findings, we formulate the identification of functional clusters as a clustering problem and propose a non-intrusive solution called *Prism*. Prism adopts a coarse-to-fine clustering strategy. It first partitions instances into coarse-grained chunks based on communication patterns. Within each chunk, Prism further groups instances with similar resource usage patterns to produce fine-grained functional clusters. Such a design reduces noises in the data and allows Prism to process massive instances efficiently. We evaluate Prism on two datasets collected from the real-world production environment of Huawei Cloud. Our experiments show that Prism achieves a v-measure of ∼0.95, surpassing existing state-of-the-art solutions. Additionally, we illustrate the integration of Prism within monitoring systems for enhanced cloud reliability through two real-world use cases.**

I. INTRODUCTION

Cloud providers such as Amazon AWS, Microsoft Azure, and Google Cloud Platform (GCP) have provided a wide range of services and ensure availability 24/7 to their customers worldwide. Guaranteeing the reliability of a cloud system is crucial since even a brief downtime could result in significant financial losses for cloud vendors and their customers [1], [2].

Cloud systems typically leverage virtualization techniques to abstract hardware resources, such as computation, storage, and networks, into instances (*e.g.*, virtual machines), serving as basic components of cloud services [3]–[5]. Such architecture provides flexibility and elasticity for tenants to subscribe various instances to run services with different functionalities *e.g.*, machine learning and database services. This, in turn, enables them to create complex and customizable applications.

However, just as each coin has two sides, such practice makes it more challenging to ensure the reliability of cloud

systems. In particular, virtualization degrades the observability of the system, *i.e.*, the ability to understand the system internal execution state. Virtualization introduces an additional layer of abstraction between the underlying hardware and the running applications, making it difficult to correlate the problems across different layers [6]. For example, an issue at the application layer may be caused by problems either within the instance itself or with the underlying hardware.

To enhance system observability, a common practice for cloud vendors is deploying a variety of monitors to collect runtime information of each instance [7]–[10], which record only data related to reliability issues without touching users' privacy. The monitoring data are then utilized for downstream maintenance tasks. For example, *communication traces*, which record network packet transmissions between instances (*e.g.*, the source and destination IP addresses and port numbers), are often used to identify abnormal network behaviors, such as network attacks and excessive traffics [11]–[15]. On the other hand, *performance metrics*, such as CPU utilization and memory usage, are commonly utilized for detecting anomalies and localizing faults [16]–[18].

The monitoring data have provided valuable insights to ensure the reliability of individual instances. However, cloud vendors still view instances as distributed black boxes without knowing how an application is deployed across the infrastructure [19]. Consequently, it can be challenging to assess the impact of issues at the platform level (such as instance or hardware problems) on applications that are deployed on top of them. For example, packet losses in individual instances are commonplace in cloud systems and are generally ignored, as they seldom impact customer applications. However, when multiple instances, all supporting the same application, concurrently experience packet losses, it likely indicates a more significant issue that users may encounter, such as interruptions due to network disconnections. The limited awareness of relationships between instances complicates the detection of such problems, thereby impeding timely mitigation efforts.

To bridge this gap and improve the system observability, we propose to infer *functional clusters* of instances, where each cluster contains the instances having similar functionalities. With this additional knowledge, cloud vendors can enhance

---

*Both authors make equal contribution to this paper.
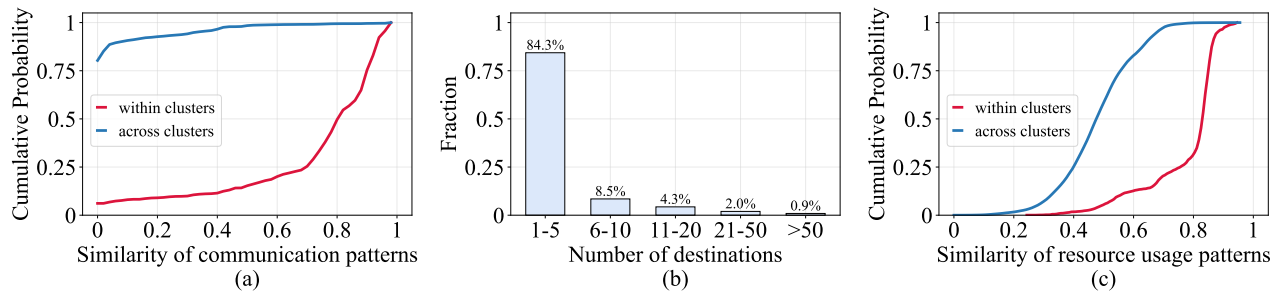**Zhuangbin Chen is the corresponding author.

the reliability of the cloud by improving various downstream management and maintenance tasks (to be detailed in §V). However, there are two major challenges that we need to overcome to achieve this goal. The first challenge is that only limited information is available. As mentioned before, cloud vendors cannot access tenants' private data, including logs and source codes. A non-intrusive solution that relies solely on external data (*e.g.*, traces and metrics) is required. The second challenge is the large scale of instances in cloud systems. A typical cloud system can consist of millions of instances in total [19], resulting in an enormous amount of data for analysis. Valuable insights are concealed within the vast and noisy data of cloud systems, making it difficult to reveal the hidden function clusters.

To tackle the first challenge and explore a feasible non-intrusive solution, we first conduct a pilot study on the services deployed in Huawei Cloud. For privacy reasons, we only use internal services of Huawei Cloud without touching tenants' instances. Specifically, we utilize a total of 3,062 internal instances covering services with 397 types of functionalities and study whether different functionalities can be identified simply based on external monitoring data. Our study uncovers that instances having similar functionalities share similar *communication* and *resource usage* patterns. Communication patterns mean that instances with similar purposes may frequently communicate with the same set of destinations, reflected in their communication traces. We find that 75% of instances within the same functional clusters have a high overlap (≥ 0.7) in their communicated destinations. Conversely, for 92% of instances with different functionalities, the overlap is only less than 0.2. Additionally, despite the large scale of instances, 99.1% of instances communicated with a limited number of destinations (fewer than 50), indicating a strong locality in communication patterns. Resource usage patterns, on the other hand, denote that instances with similar functionalities would demonstrate comparable resource consumption, which is reflected in their metrics. For example, a machine learning service is expected to exhibit greater CPU usage, while instances running an in-memory database like Redis would primarily require more memory. We find that most ($\sim$75%) of instance pairs with the same functionalities have high metric-based similarities (≥ 0.8), while the similarities decrease for those instances having different functionalities.

Motivated by the two kinds of inherent patterns of the instances, we formulate the identification of functional clusters as a clustering problem. Intuitively, we aim to cluster the instances by harmoniously integrating the communication patterns and resource usage patterns. To achieve this goal and alleviate noises within the tremendous data, we propose *Prism*, which adopts a coarse-to-fine clustering strategy. Prism consists of two components, *i.e.*, *trace-based partitioning* and *metric-based clustering*. In the trace-based partitioning step, we leverage the communication patterns to coarsely divide the entire large set of instances into smaller chunks. This step helps limit the comparison space within each chunk, thus reducing the complexity of the subsequent clustering process

Fig. 1. The hierarchical structure of cloud systems

and eliminating noises introduced by instances from other clusters. In the metric-based clustering step, we perform fine-grained clustering by comparing the resource usage patterns of instances in a pairwise manner. This step allows us to carefully group instances within the same functional cluster.

To evaluate Prism, we conduct extensive experiments on two datasets collected from the production environment of Huawei Cloud, a top-tier cloud provider serving global customers. To evaluate the generality of Prism, these datasets were procured from two regions of Huawei Cloud, each covering a diverse set of functionalities. The experimental results show that Prism achieves a v-measure of $\sim$0.95, surpassing existing state-of-the-art solutions, and is robust to parameter changes. Moreover, Prism is both scalable and efficient, with a linear time complexity, enabling it to handle a substantial number of instances. Furthermore, we have deployed Prism in Huawei Cloud, and we share two real-world use cases to demonstrate the usefulness of functional clusters in maintaining Huawei Cloud. In the first case, functional clusters showcase the ability to detect vulnerable application deployments that may be at risk of disruption due to hardware failures. The second case shows how functional clusters can aggregate minor packet loss errors across instances, thus enabling identification of latent issues that are not observable at either the instance or region level. We summarize our contributions as follows:

- We conduct a pilot study to understand the characteristics of functional clusters across over 3,000 instances based on a real-world cloud system, Huawei Cloud (§II). Our findings reveal two clues for identifying functional clusters (*i.e.*, communication patterns and resource usage patterns).
- We design a non-intrusive solution called Prism to identify functional clusters in large-scale cloud systems, which is able to effectively capture and integrate the inherent communication and resource usage patterns among instances (§III).
- Extensive experiments are conducted on two real-world industrial datasets (§IV). Our results demonstrate that Prism is effective, efficient and practically useful in identifying functional clusters in industrial cloud systems. Our dataset and code are made public to benefit the community on https://github.com/OpsPAI/Prism.

II. BACKGROUND AND PILOT STUDY

In this section, we first discuss the background of cloud systems and clarify the terminologies used. Then, we present a pilot study to understand the characteristics of instances that can facilitate the identification of functional clusters.

Fig. 2. Results of the study on communication and resource usage patterns.

A. Background

1) Cloud System Structure

Modern cloud systems are complex and highly distributed, consisting of multiple layers of hardware and software components that work together to provide on-demand computing resources to tenants. Fig. 1 shows the hierarchical structure of a typical cloud system. At the lowest layer, hardware resources such as physical machines, storage, and networking equipment form the underlying infrastructure. These resources are virtualized into environments known as *instances* (*e.g.*, virtual machines), which can be dynamically created, scaled, and terminated as needed, forming a layer of virtualization. On top of these instances, tenants can deploy services with a broad range of *functionalities* that run in different programming languages and frameworks. These functionalities can either serve as standalone *applications* or be combined to form complex *applications*. For example, an online shopping application may consist of services offering interdependent functionalities like load balancing, user authentication and databases. To ensure scalability and fault tolerance, multiple copies (or replicas) of the same service are typically created and distributed across the cloud environment to support a single functionality. This approach enables handling user traffic spikes while guaranteeing high availability of the system in case of instance failures. It is crucial to timely detect potential issues in the services with various functionalities that constitute the application in order to ensure its overall availability. This paper focuses on discovering functional clusters containing instances with similar functionalities, which are smaller and more manageable units than complex applications. With this information, operators are allowed to build more actionable monitoring metrics for safeguarding each functionality.

2) Cloud System Monitoring

Monitoring is a common practice for top-tier cloud vendors, such as AWS CloudWatch [20], Azure Monitor [21] and Google Cloud Monitoring [22]. Monitoring tools are used to collect various types of data about the system's performance and behavior. Two key types of data are commonly collected for each instance: communication traces and performance metrics. *Communication traces* data, are records of network transmissions between instances in a cloud environment. These traces are typically generated by network monitoring tools (*e.g.*, flow logs [23]–[25]) and capture metadata about the network traffic, such as the source and destination IP addresses,

port numbers, and protocol types. By collecting and analyzing these types of data, cloud system operators can take proactive measures to ensure system security and reliability [11]–[15]. *Performance metrics*, on the other hand, includes information such as memory usage and network throughput organized in the form of time series, which are used to detect and diagnose system performance issues [16], [17].

As shown in Fig. 1, tenants mostly focus on the functionalities of services they deploy, rather than delving into infrastructure-level details. On the other hand, due to privacy concerns, cloud vendors only possess runtime information about instances and hardware resources, lacking knowledge about how customers deploy services with various functionalities across these instances. While cloud providers do possess some metadata about these instances, such as which customers subscribe to particular instances, this information is often too coarse-grained. For example, thousands of instances belonging to the same enterprise customer could share the same tenant ID [19], and these instances may host a diverse range of functionalities. This paper aims to empower cloud providers with insights into more fine-grained structure of the functionalities by learning from instance data visible to them. This would facilitate building an enhanced monitoring system to improve the reliability of cloud systems (will show in §V).

B. A Pilot Study

In the following, we conduct a pilot study across over *three thousand* internal instances in Huawei Cloud, aiming to find clues to uncover the valuable functional clusters. We conduct manual inspections in collaboration with the corresponding teams within Huawei Cloud to understand their functionalities. We obtain services covering 397 types of functionalities in total, and more details about this dataset are in §IV-A.

1) Communication Pattern

The communication pattern serves as an indicator that instances within the same functional cluster tend to exhibit comparable network behaviors, as evidenced by the communication traces they generate. As inspired by [19], instances within the same functional clusters might communicate with similar destinations. To investigate this, we combine every two instances and compute the overlap of their destinations through Jaccard similarity [26]. Then, we compare the similarities within the same clusters and across different clusters.

Fig. 2-(a) presents the comparison results of communication pattern similarities within or across clusters, where we can ob-
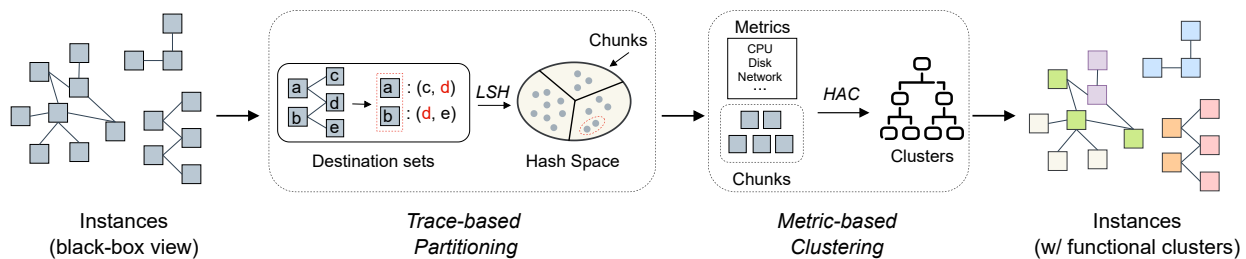
Fig. 3. The overall workflow of Prism

serve a significant difference between them. When examining instances within the same cluster, we find that 50% of the instance pairs demonstrate more than 0.8 similarity and over 75% of them exhibit more than 0.6 similarity. In contrast, when comparing instance pairs from different clusters, over 75% of the pairs exhibit a similarity score of 0, indicating no overlap between their destinations. Additionally, 96% of the pairs have a similarity score of <0.4, indicating that instances from different clusters rarely communicate with the same destinations. However, for some cross-cluster instances, there are still a little overlap in their destinations. These destinations are usually common services such as network gateway and authentication that are shared by multiple applications.

To further understand the communication patterns, we study how many different destinations one instance can frequently communicate with. Fig. 2-(b) shows the results. We can find that even though there are thousands of instances in total, the majority of the instances only communicate with a small number of destinations. For example, 84.3% of instances communicate with 1 to 5 instances, and 99.1% of instances communicate with less than 50 instances. This suggests a strong *locality* of instances, *i.e.*, most instances tend to communicate with a small set of other instances frequently.

2) Resource Usage Pattern

Intuitively, instances within the same functional cluster should observe similar patterns in their resource consumption (*i.e.*, resource usage patterns). To investigate whether resource usage patterns can be utilized to uncover functional clusters, we analyze the similarities in the metric data among instances, either within the same functional cluster or across different clusters. Thus, we compare the multivariate metric similarity on two instances using the multivariate dynamic time warping (DTW) distances [27], a distance metric to compare a pair of time series that may vary in timing (more details in §III).

Fig. 2-(c) shows the distribution of resource usage pattern similarities among instances, either within or across functional clusters. We can observe that the similarities of instance pairs within clusters are generally large, with over 75% of such pairs exhibiting 0.7 similarity or higher. In contrast, instance pairs across clusters display smaller similarities, with 92% of pairs across different clusters possessing less than 0.2 similarity. However, it is worth noting that there is a small portion ($\leq 10\%$) of cross-cluster instance pairs that have high metric-based similarities, with a value of ≥ 0.8. This is reasonable since instances having different functionalities could behave similarly, *e.g.*, have a high CPU utilization. Nevertheless, it

still suggests that leveraging the similarities between instance metrics is promising in distinguishing their functional clusters.

Summary. We summarize our findings as follows.

- Instances that belong to the same functional cluster exhibit comparable communication patterns, as evidenced by the considerable overlap in their communication destinations. Furthermore, the analysis reveals that the majority of instances interacted with a limited number of other instances, indicating a strong locality of instances.
- Instances within clusters generally exhibit high similarities in their resource usage patterns, while instance pairs across clusters show smaller similarities.
- While communication and resource usage patterns provide valuable insights, they are not entirely reliable indicators for distinguishing between different functional clusters, as some *noises* in the form of cross-cluster instances with high similarities in both patterns are observed.

III. METHODOLOGY

A. Overview

The goal of this paper is to design a non-intrusive solution to discover functional clusters among massive instances in a large-scale cloud system. The input is an entire set of instances and their associated monitoring data, *i.e.*, communication traces and performance metrics. The output of our approach is multiple clusters, where each cluster represents a functional cluster consisting of instances that have similar functionalities.

To achieve this goal, we propose Prism, an automated approach that can effectively discover functional clusters based on both the communication patterns and resource usage of instances. Fig. 3 illustrates the overall workflow of Prism, which comprises two main components: *trace-based partitioning* and *metric-based clustering*. Given a set of instances, Prism adopts a two-stage clustering process, which progressively divides the entire set of instances to coarse-grained *chunks*, then fine-grained *functional clusters*. Specifically, the *trace-based partitioning* step is inspired by the strong locality of communication patterns, as shown in §II-B1. Based on communication patterns, Prism first separates all instances into different chunks. Instances in the same chunk share similar communication destinations. By dividing the complete instances set into multiple small chunks, we can reduce the noises introduced from other instances during the subsequent fine-grained clustering step. For each chunk, *metric-based clustering* is then applied to generate fine-grained clusters by measuring the similarities of monitoring metrics of instances. Finally, instances belonging to the same resultant cluster are

considered to have similar functionalities. Such a coarse-to-fine design avoids pairwise comparisons between a large number of instances and reduces noises between instances, making Prism salable and practical for large-scale cloud systems.

It is important to note that Prism relies solely on external monitoring data and does not access any of the tenants' private data, which ensures that there are no privacy concerns. While we can infer which instances have similar functionalities, we cannot identify the specific type of the functionalities in use. This approach maintains our tenants' confidentiality.

B. Trace-based partitioning

As studied in §II-B1, instances sharing the same functional clusters are more likely to communicate with a similar set of destination hosts. Thus, the trace-based partitioning of Prism measures the communication pattern similarity and divides instances into coarse-grained *chunks*.

Data Preprocessing. Let x_i represent an instance in the cloud system. Communication traces can be represented as tuples of the form (x_{src}, x_{dst}), where x_{src} and x_{dst} represent the instances that communicate with each other. By analyzing the communication traces, we can obtain the *destination set* of each instance, denoted by $S_i = (x_1, x_2, x_3, ...)$, which contains all the instances that have communicated with x_i. However, as demonstrated in §II-B1, instances with dissimilar functionalities may share common destinations, such as network gateways, which can introduce noise when comparing the communication patterns between instances. To mitigate this issue, we remove instances that interact with more than 100 different instances, which is rare as shown in Fig. 2-(b).

Jaccard Similarity-based Partitioning. Next, we divide all instances into chunks by measuring how much their destination sets overlap. To achieve this, a straightforward solution is to calculate the Jaccard similarity [28] of destination sets of every pair of instances, which is denoted as $J(x_i, x_j) = \frac{|S_i \cap S_j|}{|S_i \cup S_j|}$, *i.e.*, the ratio of the size of their intersection to the size of their union. However, it requires conducting pairwise comparisons between millions of instances in a large-scale cloud system. This process can be extremely time-consuming and may render the approach unfeasible in practice.

To address this issue, we propose to leverage locality-sensitive hashing (LSH) [29] to enable efficient partitioning. LSH is a technique developed for identifying similar items in large datasets. Its idea involves hashing the items into signatures such that similar items are more likely to be assigned to the same bucket. Given a query, LSH can efficiently return similar items with a sub-linear time cost without pairwise comparison with the entire instance set. In our context, we combine LSH with the MinHash function, which allows items with high Jaccard similarities put into the same buckets [30].

Algorithm 1 describes the trace-based partitioning process. First, we extract the destination sets S of each instance from historical communication traces (lines 1-5). Second, for each instance x_i, we apply MinHash function to its destination set S_i to obtain the hash signature. The hash signature is then inserted into the LSH model (lines 7-10), which assigns

---

Algorithm 1: Trace-based Partitioning

Input: List of instances: $\mathcal{X} = \{x_1, x_2, ..., x_l\}$;
　　　　Communication trace records: $\mathcal{R} = \{r_1, r_2, ..., r_t\}$;
　　　　Similarity threshold: θ_{LSH}
Output: Multiple instance chunks: $\mathcal{C} = \{C_1, C_2, ...\}$
Init: $S \leftarrow$ Empty list of feature sets; $M_{LSH} \leftarrow$ empty LSH
　　　model; $U \leftarrow$ Disjoint-set data structure

1 // (1) Construct feature sets
2 **for** $i \leftarrow 1$ **to** t **do**
3 　　$x_{src}, x_{dst} \leftarrow r_i$
4 　　$S[x_{src}]$.insert(x_{dst})
5 **end**
6 // (2) Build the LSH model
7 **for** *each instance* $x_i \in \mathcal{X}$ **do**
8 　　$S_i \leftarrow S[x_i]$
9 　　M_{LSH}.insert(MinHash(S_i))
10 **end**
11 // (3) Search neighbors and build chunks
12 **for** *each instance* $x_i \in \mathcal{X}$ **do**
13 　　$S_i \leftarrow S[x_i]$
14 　　$\mathcal{N}_i = M_{LSH}$.search(S_i, θ_{LSH}) // find neighbors
15 　　**for** *each instance* $x_j \in \mathcal{N}_i$ **do**
16 　　　　**if** U.findSet(x_i) != U.findSet(x_j) **then**
17 　　　　　　U.unionSet(x_i, x_j) // merge x_i and neighbors
18 　　　　**end**
19 　　**end**
20 **end**
21 $\mathcal{C} \leftarrow$ U.getAllSets()

---

the item to a bucket. Third, for each instance x_i, we search its nearest neighbors $\mathcal{N}_i$ within the buckets produced by the LSH model (lines 12-14). Here, a manual-defined threshold $\theta_{LSH} \in [0, 1]$ is included, where a smaller θ_{LSH} value allows more dissimilar neighbors to be included. After that, we group the instance x_i with its neighbors $\mathcal{N}_i$ based on the Disjoint-set data structure U (lines 15-19). This data structure U provides two efficient operations, *i.e.*, $U.findSet$ that find the set that contains a specific item and $U.unionSet$ that merge two disjoint sets. If we find the sets containing x_i and containing x_j are disjoint (line 16), we merge these two sets (line 17) since x_i and x_j are similar. In this way, we progressively divide the entire set of instances into multiple disjoint sets (*i.e.*, chunks) managed by U. Finally, we can obtain all the instance chunks $\mathcal{C}$ by enumerating the records in U (line 21).

The trace-based partitioning algorithm is highly efficient for two reasons. First, we bypass the expensive pairwise similarity computation for all the instances by using LSH with MinHash. Secondly, we leverage the disjoint-set data structure to merge similar instances into chunks efficiently. The *findSet* and *unionSet* operations of the disjoint-set data structure can be completed within nearly constant time complexity, which further ensures the efficiency of the merging process. Moreover, the number of neighbors $\mathcal{N}_i$ (line 14) is generally fewer than 50, which is much smaller than the total instance number $\mathcal{X}$ (line 12) due to the locality of communication patterns (Fig. 2-(b)), which improves Prism's scalability, making it feasible for large-scale cloud systems like Huawei Cloud.

C. Metric-based Clustering

Trace-based partitioning tends to group as many instances as possible together, which can inevitably include instances

with different functionalities to the same chunk. The reason is that instances from different clusters can still communicate to the same destinations (as studied in §II-B1), and this leads to overlap of the destination sets of these instances, which may be wrongly grouped together.

To address this problem, we further group these instances by utilizing more fine-grained monitoring metrics that record detailed runtime information of instances (*i.e.*, resource usage patterns as studied in §II-B2). Each instance is monitored via multiple dimensions to ensure its reliability, producing multivariate metrics, including CPU utilization rate, network incoming/outgoing bytes rate, disk read/write request rate, and disk read/write bytes rate. In the following, we aim to calculate a metric-based distance for each pair of instances. Then, we can cluster those instances that are close to each other.

Data Preprocessing. We apply the following preprocessing techniques to the raw metric data collected to remove noises and normalize the data within a comparable scale. First, we regard apparent extreme values as anomalous noises within the metric data because these values can bias the subsequent distance computation step. For each metric, we replace the data points that are out of the three-sigma range with the average value of the nearest ten points. Next, since the amplitude scales of different metrics are different, *e.g.*, network-related metrics are highly variable and may range from tens of bytes to millions of bytes. This can make the produced distances incomparable between instances with different network traffic volumes. To address this issue, we apply natural logarithm to these metrics following [19] to make it more robust to its variance. The logarithm only solves the issue of highly variable amplitudes but does not ensure that the data points fall within the same range. Therefore, finally, we apply min-max normalization to scale each of the metrics to the range of 0-1, allowing comparison across different metrics. Formally, using y to denote a metric time series, the normalized values can be calculated as $y' = \frac{y - min(y)}{max(y) - min(y)}$.

Metric-based Distance Calculation. For an instance x, its preprocessed monitoring metrics form a group of multivariate time series represented as a matrix $\mathbf{M}_i \in \mathbb{R}^{n \times k}$, where n is the number of timestamps and k is the number of metrics used. We measure the metric-based similarity of two instances using a distance that simultaneously considers all the multivariate metrics of them. To achieve this, we first compare each metric, then aggregate the distances to produce an overall distance.

Specifically, we adopt dynamic time warping (DTW) distances [27] for distance measurement. The reason we use DTW is to overcome the problem that the monitoring metrics of different instances can have time shifts, namely, these time series may not be aligned in terms of the collection timestamps, making traditional distance measures such as Euclidean distance ineffective. In contrast, DTW allows for flexible matching of similar patterns in the time series, even when they occur at different timestamps. Based on the DTW calculation, the overall distance $d(x_i, x_j)$ between two instances x_i and

x_j can be formulated as follows:

$$d(x_i, x_j) = \sum_{u=1}^{k} \omega(i,j)_u \times DTW\big(\mathbf{M}_i(:, u), \mathbf{M}_j(:, u)\big), \quad (1)$$

$$\omega(i,j)_u = \frac{\omega(i,j)'_u}{\sum_{v=1}^{k} \omega(i,j)'_v}, \quad (2)$$

$$\omega(i,j)'_u = \frac{1}{2}\big(\sigma(\mathbf{M}_i(:, u)) + \sigma(\mathbf{M}_j(:, u))\big), \quad (3)$$

where u denotes the metric in concern, $\mathbf{M}_{i/j}(: .u)$ is the u_{th} column of the corresponding metric matrix. In particular, we use $\omega(i,j)_u$ as a weight associated with the u_{th} metric to measure the importance of each metric. Each weight is calculated as the average of the standard deviation (*i.e.*, $\sigma(\cdot)$) of the two metrics of corresponding instances as shown in Equation 3, which is normalized to the range of 0 to 1 across different metrics using Equation 2. In doing this, we reduce the weight of the metrics that barely fluctuate (*e.g.*, two instances keep the CPU utilization rate around 80%), since these metrics are less informative in representing the characteristics of instances. In contrast, if two metrics are simultaneously changing following the same trend, they are more likely to indicate instances performing the same functionalities.

Clustering Algorithm. We then apply a clustering algorithm in each *chunk* based on the metric-based distances to produce more fine-grained clusters (*i.e.*, functional clusters). Specifically, we choose the hierarchical agglomerative clustering (HAC) [31] algorithm because it allows us to adjust the number of produced clusters via setting a distance threshold, *i.e.*, θ_{HAC}. The clustering algorithm starts by considering each instance as a single cluster and then iteratively merges the closest pairs of clusters until a user-defined threshold θ_{HAC} is reached. In this process, we use complete linkage [32] to find the closest pair of clusters, *i.e.*, the distance between two clusters is defined as the maximum DTW distance between any pair of instances in the two clusters.

While HAC requires the computation of distances between instances in a pairwise manner, it is still efficient since HAC is applied separately in each chunk. Recall that chunks are produced by the trace-based partitioning step, and each chunk only contains tens of instances because of the locality of communication patterns (as shown in §II-B1). Therefore, the computation within each small chunk can significantly reduce the computation cost, making our framework scalable to a large number of instances in cloud systems.

IV. EVALUATION

We evaluate Prism by answering the following research questions (RQs):

- **RQ1:** How effective is Prism in clustering instances having similar functionalities?
- **RQ2:** How does each component contribute to the overall performance of Prism?
- **RQ3:** What is the parameter sensitivity of Prism?
- **RQ4:** What is the efficiency of Prism?

Datasets	# Functionalities	# Instances	# Traces	# Metrics
Dataset $\mathcal{A}$	292	2,035	100.2 M	7.25 M
Dataset $\mathcal{B}$	105	1,027	121.6 M	3.71 M
Total	397	3,062	212.6 M	10.96 M

A. Experimental Setup

Dataset. We evaluate Prism using two datasets collected from the production environment of Huawei Cloud. To evaluate the generalizability of Prism, the two datasets ($\mathcal{A}$ and $\mathcal{B}$) are collected from two different geographically isolated regions with different numbers of users. The detailed statistics of the two datasets are listed in Table I. These datasets only include instances that are subscribed by internal customers, where we are able to manually inspect their functionalities by collaborating with corresponding teams. We select the instances running on our production environment that are most frequently invoked according to their communication traces. Then, we reach the owners of these instances to figure out the concrete functionalities these instances support, and we finally obtain 3,062 labeled instances. Although we are unable to fully cover all instances within the Huawei Cloud due to the manual effort required, our datasets encompass a diverse range of functionalities (397 types in total), such as databases, disaggregated memory, authentication servers, search engines, and machine learning algorithms. Such diversity would help evaluate whether a clustering algorithm can generalize to different functionalities. Additionally, these functionalities can belong to different applications. For example, while various applications may each have their own databases, these database functionalities are distinguished from one another in our datasets since they are utilized by distinct applications that serve diverse workloads (e.g., databases of an online shopping application and a face recognition application). For the monitoring data, traces are extracted from the network packet transmission records, while metrics are collected at five-minute intervals. Given the extensive usage and frequent communication of instances, we ultimately collect hundreds of millions of traces. In terms of metrics, the total number of points is 10.96 million for all instances. We have made our datasets publicly available in our GitHub repository. However, due to confidentiality concerns, the actual functionality names have been anonymized and are represented as "cluster_ID".

Evaluation Metrics. We use the metrics *homogeneity*, *completeness* and *V-measure* to evaluate the effectiveness of Prism in grouping the instances within the same functional cluster. These metrics have been widely adopted in evaluating the quality of clustering results in previous studies. Homogeneity measures the proportion of instances in the same cluster that share the same ground truth labels. Completeness, on the other hand, measures the proportion of instances with the same ground truth labels that are grouped into a single predicted cluster. V-measure is a harmonic mean of homogeneity and completeness, providing an overall indicator for clustering per-

Methods	Dataset $\mathcal{A}$			Dataset $\mathcal{B}$		
	Homo.	Comp.	V Meas.	Homo.	Comp.	V Meas.
OSImage	0.238	0.894	0.376	0.258	0.889	0.400
CloudCluster	0.346	0.748	0.473	0.369	0.753	0.495
ROCKA	0.831	0.882	0.856	0.875	0.900	0.887
OmniCluster	0.932	0.862	<u>0.896</u>	0.944	0.877	<u>0.909</u>
Prism	0.976	0.916	**0.945**	0.979	0.922	**0.950**

formance considering the trade-off between these two metrics.
Competitors. We select the competitors from recent studies:

- *OSImage* is a basic baseline that uses the name of the operating system (OS) image to differentiate between instances. Cloud providers offer various pre-installed OS images to cater to diverse customer needs. For example, an OS image named *deeplearning-pytorch-2.0* implies that the instance is designed for executing deep learning applications.
- *CloudCluster* [19] clusters instances based on their pairwise traffic matrix in cloud projects to determine the functional structure of the cloud service. It normalizes each row of the traffic matrix by feature scaling, then reduces its dimensionality through low-rank approximation. Finally, HCA is employed to group all instances.
- *ROCKA* [33] aims to cluster instances by using their monitoring metrics. ROCKA first normalizes the metrics to eliminate amplitude differences. It then uses shape-based distance (SBD) as a distance measure, which is robust to phase shift and efficient for high-dimensional time series data. Then, clusters are created based on DBSCAN algorithm.
- *OmniCluster* [34] clusters instances based on multivariate metrics of each instance. It employs a one-dimensional convolutional autoencoder (1D-CAE) to extract the low-dimensional features of all metrics. These features are selected based on their periodicity and redundancy. Finally, it uses HAC to divide all instances into different clusters.

B. RQ1: Effectiveness in functional cluster Discovery

In this RQ, we evaluate the accuracy of the functional clusters discovered by Prism in comparison with state-of-the-art baseline methods. To achieve this, we apply Prism and baseline methods to cluster instances in the dataset of $\mathcal{A}$ and $\mathcal{B}$. We present the results of our experiments in terms of homogeneity (Homo.), completeness (Comp.), and v-measure (V Meas.) in Table IV-A, where we highlight the best V Meas. with boldface and the second-best ones with underline.

It can be observed that Prism outperforms three state-of-the-art baseline methods, namely CloudCluster, ROCKA, and OmniCluster, by a significant margin, achieving V-measures of 0.945 and 0.950 on datasets $\mathcal{A}$ and $\mathcal{B}$, respectively. These results indicate that Prism can achieve the best balance between homogeneity and completeness. This can be attributed to the fact that Prism effectively integrates communication and resource usage patterns to discover functional clusters. Unlike Prism, baseline methods typically focus on either trace or metric data, leading to worser performance. Specifically, OSImage exhibits low homogeneity but high completeness, as using only image names to separate instances can overly group instances

TABLE III
CONTRIBUTION OF DIFFERENT COMPONENTS IN PRISM

Methods	Dataset $\mathcal{A}$			Dataset $\mathcal{B}$		
	Homo.	Comp.	V Meas.	Homo.	Comp.	V Meas.
Prism	0.976	0.916	**0.945**	0.979	0.922	**0.950**
Prism w/o Metrics	0.462	**0.920**	0.615	0.463	**0.949**	0.622
Prism w/o Traces	0.949	0.869	<u>0.907</u>	0.915	0.893	<u>0.904</u>

with different functionalities that share the same images. While CloudCluster outperforms OSImage in v-measure, it falls short of other metric-using baseline methods, suggesting that metric similarities are more effective in distinguishing functionalities than communication trace similarities.

Answer to RQ1: Prism outperforms all state-of-the-art comparative methods in revealing the functional clusters across two different datasets, achieving a v-measure of 0.945 and 0.950 in dataset $\mathcal{A}$ and $\mathcal{B}$.

C. RQ2: Contribution of Each Component

In this RQ, we evaluate each component's contribution to Prism's overall performance. We created two Prism variants and compared them with the original approach across datasets $\mathcal{A}$ and $\mathcal{B}$. The first, *Prism w/o metrics*, eliminates metric-based clustering, relying solely on communication destination similarity. The second, *Prism w/o traces*, omits trace-based partitioning, directly applying the HAC algorithm to cluster instances based on resource usage patterns.

We present the comparison results in Table IV-B, from which we make the following observations. (1) Removing either of the two components can adversely affect the performance of Prism, underscoring the necessity of integrating both communication and resource usage patterns. (2) The V-measure of *Prism w/o metrics* is significantly lower than that of *Prism* and *Prism w/o traces*, primarily due to its low homogeneity. This suggests that the trace-based partitioning step over-clusters many instances that should be separated. The communication pattern alone is not distinctive enough because instances having different functionalities should still communicate with some common instances, such as network gateway and proxy services (as illustrated in Fig. 2-(a)). Nonetheless, the use of solely communication patterns achieves the best completeness score, implying that it barely separates clusters that should be grouped. (3) *Prism w/o traces* has the lowest completeness score, indicating that it can overly split clusters apart, but it has a considerably high homogeneity. This observation implies that Prism harnesses the benefits of both performance metrics and communication traces, achieving the optimal balance between homogeneity and completeness.

Answer to RQ2: The variants, *Prism w/o metrics* and *Prism w/o traces*, each sacrifice either homogeneity or completeness. Yet, Prism effectively combines communication traces and metric data, yielding the highest v-measure, *i.e.*, a balanced performance in completeness and homogeneity.

D. RQ3: Parameter Sensitivity

In the design of Prism, we identify the following two parameters that are manually selected and potentially affect the

(a) Impact of θ_{LSH} (b) Impact of θ_{HAC}

Fig. 4. Parameter Sensitivity of Prism

performance of Prism. For clarity, we present the evaluation results in Dataset $\mathcal{B}$; similar results are obtained in dataset $\mathcal{A}$.

1) LSH threshold (θ_{LSH})

In §III-B, we utilize LSH algorithm to perform a search for similar neighbors during the trace-based partitioning step. The LSH algorithm groups similar items together into the same bucket with high probability, but it cannot guarantee that all items in the same bucket are actually similar; therefore, θ_{LSH} is utilized to filter dissimilar items within each bucket.

We varied the value of θ_{LSH} from 0 to 1 with a step size of 0.1 and evaluated the performance of Prism. The results, shown in Fig. 4(a), indicate that the V-measure remains stable with only a slight decrease as θ_{LSH} increases, which is primarily due to the decrease in completeness. This is because the LSH algorithm has already grouped similar items together into different buckets. Furthermore, since the communication patterns of most instances are distinct from one another, as depicted in Fig. 2-(a), there are only a small number of dissimilar items in the same bucket. As a result, adjusting θ_{LSH} does not significantly affect the clustering results.

2) HAC threshold (θ_{HAC})

In §III-C, HAC is used for clustering instances within each chunk, where the parameter θ_{HAC} controls the granularity of clustering: a smaller value of results in more fine-grained clusters, while a larger value results in fewer, coarser clusters.

We enumerated the value of θ_{HAC} from 0 to 1 with a step size of 0.1 and evaluated the performance of Prism. The results are shown in Fig. 4(b). We observed that increasing θ_{HAC} can increase completeness and decrease the homogeneity. This is because larger clusters are generated when θ_{HAC} is larger. The best v-measure is achieved when θ_{HAC} is around 0.4. Subsequently, there is a slight decrease in homogeneity, while the v-measure remained stable. This decline in homogeneity is due to the inclusion of more dissimilar instances in a cluster, thereby reducing its homogeneity. Nevertheless, the preceding trace-based partitioning step groups similar instances together, resulting in a limited number of dissimilar instances. Hence, the overall performance is not significantly affected.

Answer to RQ3: Prism is not significantly sensitive to the parameters θ_{LSH} and θ_{HAC}. This is because the trace-based partitioning step already groups similar instances together and separates dissimilar instances based on their communication patterns. Thus, adjusting these two parameters only has a minor effect on the clustering results.

Methods	# Instances				
	1,000	5,000	10,000	50,000	100,000
CloudCluster	0.9	23.87	78.65	1768.7	5585.7
ROCKA	80.7	1981.8	7850.3	-	-
OmniCluster	31.7	264.6	1048.6	26531.8	-
Prism w/o Metrics	3.9	19.1	40.2	195.1	392.4
Prism w/o Traces	80.3	2066.1	8232.3	-	-
Prism	18.2	89.4	183.9	929.2	1912.7

E. RQ4: Efficiency of Prism

In this section, we assess the efficiency of Prism in the context of large-scale cloud systems with millions of instances that are frequently created, deleted or updated. To this end, we apply them to 1,000 / 5,000 / 10,000 / 50,000 / 100,000 instances and record the time needed (in seconds) to complete the clustering process.

Table IV-E presents the results, from which we can make the following observations: (1) ROCKA, OmniCluster, and Prism w/o Traces require increasingly more time as the number of instances increases, and they cannot complete the clustering process within a reasonable time when clustering 100,000 instances. This is mainly because these methods require pairwise similarity computation based on instance metrics, resulting in a quadratic growth in time complexity as the number of instances increases. OmniCluster mitigates this issue by reducing the dimensionality of metrics, requiring less time than the other two methods. (2) CloudCluster and Prism w/o Metrics are more efficient than other baseline methods. Prism w/o Metrics is more efficient because we optimize efficiency using pair-wise comparison with LSH and MinHash, as described in §III-B. (3) Prism is less efficient than Prism w/o Metrics since it requires an additional metric-based clustering step. In addition, when the number of instances is fewer than 10,000, CloudCluster outperforms Prism because the time required by Prism to build the LSH index is dominant. However, as the number of instances increases to 100,000, Prism's efficiency becomes superior to other baselines, being four times faster than CloudCluster. This is attributed to the coarse-to-fine clustering process, which limits pairwise distance computation within small chunks. Therefore, the time cost of Prism only increases linearly with the instance numbers.

Answer to RQ4: Compared with state-of-the-art solutions, Prism is the most efficient solution when processing a large number of instances (*e.g.*, 100,000). Moreover, thanks to the coarse-to-fine strategy of Prism, its time cost increases linearly with an increasing number of instances, making it scalable for handling massive instances in cloud systems.

V. INDUSTRIAL EXPERIENCE

In this section, we share our experience in applying Prism to a real-world cloud system (*i.e.*, Huawei Cloud), which aims to demonstrate the practical usefulness of Prism. Generally, customers usually subscribe instances from Huawei Cloud in a batch manner, *e.g.*, thousands of instances. These customers
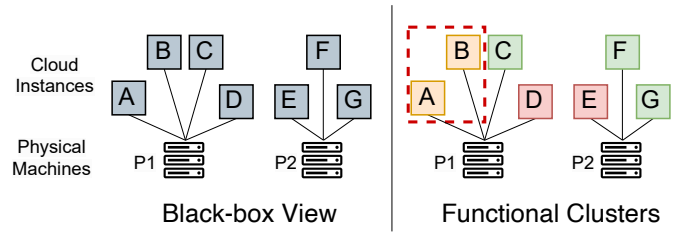

Fig. 5. Case I: vulnerable deployment identification

can then concentrate on the development and deployment of a variety of services across these instances, while the cloud providers handle the often tedious tasks of maintenance and operation to ensure system reliability. Due to privacy concerns, on-site engineers from Huawei Cloud can only rely on limited runtime information of these instances (*e.g.*, network packet drop rate) to monitor their health states [8], [10], [35]. However, without knowing how customers' applications are organized in these instances, we observe that some potential threats in the deployment or underlining errors may be missed, which may later cause service interruptions, consequently impacting the overall availability of the deployed applications [19]. To address this problem, in Huawei Cloud, we adopt Prism to reveal functional clusters in the massive instances hosted by Huawei Cloud. These functional clusters provide additional information regarding the structure of service deployment across the instances, thus enabling us to conduct more comprehensive and fine-grained monitoring of the cloud system. We present two primary usage scenarios of functional clusters within Huawei Cloud: *vulnerable deployment identification* and *latent issue discovery*.

A. Vulnerable Deployment Identification

Functional clusters can help cloud providers identify instances with vulnerable deployments. Specifically, a vulnerable deployment refers to a scenario where all instances, having the same functionalities, are deployed on the same physical machines. In such case, once a failure happens on this physical machine (*e.g.*, disk failure [36]), the entire functionality can be interrupted. In contrast, if these instances were distributed across different physical machines, only a subset of the instances would be affected in the event of a failure, thereby preventing a complete shutdown of the functionality. However, due to the abstraction of physical resources into instances, customers often deploy their applications within these instances without understanding how these instances are distributed across actual physical machines. On the other hand, cloud vendors possess knowledge of the mapping between instances and physical machines; yet, they are often unaware of the organization of functionalities across these instances due to privacy concerns. Given the vast number of instances in a cloud system, manually identifying vulnerable deployments poses a significant challenge for on-site engineers.

To fill in this gap, we apply Prism to identify functional clusters to help detect potentially vulnerable deployments. Fig. 5 provides a concrete example. The left-hand side presents a black-box view of instance deployment from a cloud ven-
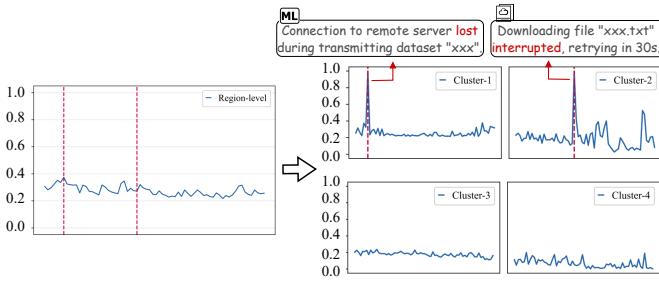
Fig. 6. Case II: latent issue discovery

dor's perspective, where only the information about which instances are deployed on which physical machines is known. In contrast, the right-hand side displays instances with functional clusters. With this knowledge, we can identify three functionalities: a functionality including A and B (marked as yellow), a functionality including D and E (marked as red), and a functionality including C, F, and G (marked as green). The deployment of the yellow functionality is potentially vulnerable because both A and B are deployed on physical machine P1. In contrast, the other two functionalities are more reliable since their instances are deployed across two different physical machines, making them resilient to the failure of either machine. It is worth noting that although Prism can hardly pinpoint what specific functionality of an instance serves, it can identify the instance group having the same functionalities, which facilitates automatic vulnerable deployment identification without violating privacy policies.

We have applied Prism in Huawei Cloud to discover functional clusters for around 3,000 internal instances and identified *eight* cloud services with vulnerable deployments. We then contacted the corresponding teams, confirmed the existence of the vulnerable deployments, and assisted in migrating the instances across different physical machines for improved resilience. In the future, our goal is to broaden the adoption of Prism to benefit a wider group of users and help enhance the reliability of their application deployment.

B. Latent Issue Discovery

The second typical use case of Prism in Huawei Cloud is to identify latent network issues that may not be discovered by traditional monitoring methods. Modern cloud providers have been equipped with various monitoring tools to ensure the quality of their network services (*e.g.*, flow logging of AWS [24]). It is essential for such monitoring tools to comprehensively discover underlining problems in the cloud systems that can affect user experience, but without firing too many false alarms to distract the on-site engineers.

One crucial type of network monitoring is to monitor the packet loss of each instance. Packet loss, which denotes network packets that are accidentally dropped, can usually occur in any instance of a cloud system. However, they may not necessarily indicate a problem, as these errors could be caused by transient network congestion and may not affect users' experience. Considering the vast number of instances in a large-scale cloud system, a significant number of packets could be lost every minute. This presents a challenge for cloud

vendors in converting this fragmented packet loss data into actionable alarms for on-site engineers.

To address this problem, we resort to the aggregation of packet loss data from a selected group of instances, using an appropriate granular approach to identify potential problems. The underlying assumption here is that *simultaneous* packet losses occurring within a group of instances are more likely to impact user applications. For instance, if all instances within a region experience packet loss within a short time frame, it strongly suggests a regional network issue. However, one large region can contain millions of instances, and consequently, grouping by a region might fail to reveal local issues for a particular application. Another possible solution is to utilize the metadata (*e.g.*, the TenantID of the customer) to group instances. Nonetheless, there could still be tens of thousands of instances associated with the same identifiers [19]. For example, all instances subscribed to by the same enterprise customer would share the same identifier.

Prism enables a more effective approach, which is to aggregate lost packets in the granularity of the (approximated) functional clusters, which can reveal latent issues that may not be visible at neither a coarser level (*e.g.*, regional level) nor a finer level (*e.g.*, instance level). Fig. 6 shows the changes in the number of lost packets (normalized) calculated at either the region grain (left-hand side) or functionality grain (right-hand side). We can observe that while the numbers of packet loss barely change for the whole region, some functionalities (*i.e.*, Cluster-1 and Cluster-2) experience sudden increases in packet loss. This indicates that there may be latent issues affecting the performance of those specific functionalities, which are unnoticed if monitored at the region level. We then contact the corresponding teams and confirm that Cluster-1 and Cluster-2 correspond to machine learning and storage functionalities, respectively. We then validate these latent issues, and both functionalities experience interruption due to unstable network states, as evidenced in their log messages shown in Fig. 6. This highlights the potential of Prism in facilitating identifying issues that customers are experiencing without accessing their private data, which allows cloud vendors to provide more comprehensive monitoring to enhance the reliability of the cloud systems.

Enhanced Cloud Monitoring Based on Prism. To summarize, these two use cases demonstrate that functional clusters can be utilized with existing monitoring tools and enable identifying vulnerable deployment and discovering latent issues automatically. Prism plays a crucial role to provide comprehensive and precise functional clusters for large-scale instances. With the significant growth of modern cloud systems, instances experience frequent dynamic changes, including creation, deletion, and migration. In this context, Prism can be utilized to efficiently capture relations between instances. Unlike using pre-defined and rule-based monitoring [35], [37], Prism is adaptive to the frequent evolution of cloud applications. By continuously monitoring metrics like packet loss and the distribution of instance deployments, the monitoring system can effectively detect anomalies, such as sudden spikes

in packet loss or scenarios indicating vulnerable deployments. This enables prompt alerts to the on-site engineers of relevant teams, resulting in shorter response time and more efficient issue resolution. Overall, the effectiveness and efficiency of Prism significantly contribute to improving the overall monitoring and management of instances in modern cloud systems.

VI. THREATS TO VALIDITY

External Validity. The primary external threat of this study is the investigated object. The datasets are collected from Huawei Cloud, as there are no publicly available datasets that include both instance data and corresponding functionality labels. However, Huawei Cloud is a world-leading cloud provider with a vast scale. The data collected from the production environment records real behaviors of instances and covers a broad range of functionalities from two large regions as detailed in §IV-A. Therefore, the Huawei Cloud evaluation is representative and convincing. The data used by Prism, which includes traces and metrics, is typically collected by modern cloud vendors like AWS [20] and GCP [22]. This suggests that our solution could be applied to similar cloud systems, potentially benefiting cloud customers globally.

Internal Validity. The primary internal factors that could potentially compromise validity are implementation and parameter setting. To address the implementation threat, we closely followed the original papers for baseline approaches that lacked open-sourced code and re-implemented them accordingly. To minimize this threat further, we utilized several mature libraries (*e.g.*, scikit-learn) for implementing the core algorithms. Moreover, both our proposed methods and the baseline methods were subject to rigorous peer code review. To mitigate the parameter setting threat, we fine-tuned the baseline methods utilizing a grid-search approach, subsequently selecting the most optimal results.

VII. RELATED WORK

A. Instance Clustering

Communication traces are usually modeled as a communication graph for clustering instances. Xu et al [14] perform network-aware clustering for end hosts with the same network prefixes by using bipartite communication graphs. [11] [12] [13] attempt to mine the pattern of instance-to-instance communication, then detect abnormal traces to safeguard the instances. Pang et al. [19] propose CloudCluster, which uses a novel combination of feature scaling, dimensionality reduction, and hierarchical clustering to cluster a large scale of instances. Another line of work utilizes various technologies to model multivariate metric data of instances. For example, Kane et al. [38] employ Principal Component Analysis (PCA) to transform multivarite metric data to univariate time series before clustering. The most recent work, ROCKA [33] adopts shape-based distance (SBD) [39] as a robust distance measurement for clustering. OmniCluster [34] adopts hierarchical agglomerative clustering to cluster instances represented by low-dimension representations. Contrary to previous studies,

Prism effectively combines communication traces and multivariate metric data, surpassing state-of-the-art solutions by utilizing both data types, as shown in Section IV.

B. Reliability of Cloud Systems

Extensive efforts have been made to examine and understand the important factors contributing to cloud system reliability. For example, Chen et al. [7] extensively studied large-scale public cloud incidents, analyzing disruptions and failures to pinpoint reliability issues. Similarly, Huang et al. [40] explored the effects of gray failures in cloud systems. In addition, other studies [9], [36], [41], [42] reviewed public or internal postmortem reports and summarized the causes of outages in cloud systems. These studies underscore the need for enhanced understanding and observability of interdependent cloud system components. Furthermore, researchers have explored automated solutions for timely incident detection [35], [37], [43], [44] and failure mitigation [6], [10], [45]–[47] in cloud systems. Despite the promising results of these solutions in improving cloud system reliability, most [6], [44], [47] rely on well-abstracted incident descriptions and clear system topologies, often unavailable or incomplete to cloud providers [10]. In contrast, Prism's functional clusters can supplement these methods and provide insights to enhance these tasks, as shown in §V.

VIII. CONCLUSION

This paper presents an approach to enhance the observability of cloud systems by inferring functional clusters of instances. To achieve this, we conduct a pilot study based on the real-world datasets collected in Huawei Cloud, indicating that communication patterns and resource usage patterns are two essential indicators for revealing functional clusters. Motivated by our findings, we propose a non-intrusive, coarse-to-fine clustering method, Prism, which effectively integrates both communication and resource usage patterns. Experiments on two industrial datasets are conducted to evaluate Prism. Our results show that Prism outperforms state-of-the-art solutions with a v-measure of 0.95; and Prism can efficiently process massive instances. Furthermore, we share our experiences in applying Prism in Huawei Cloud. Two cases, *i.e.*, vulnerable deployment identification and latent issue discovery, demonstrate the usefulness of Prism in improving the reliability of Huawei Cloud.

IX. ACKNOWLEDGEMENT

REFERENCES

[1] Statista, "Average cost per hour of enterprise server downtime worldwide in 2019," 2020. [Online]. Available: https://www.statista.com/statistics/753938/worldwide-enterprise-server-hourly-downtime-cost/

[2] J. Chen, X. He, Q. Lin, Y. Xu, H. Zhang, D. Hao, F. Gao, Z. Xu, Y. Dang, and D. Zhang, "An empirical investigation of incident triage for online service systems," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2019, pp. 111–120.

[3] N. Jain and S. Choudhary, "Overview of virtualization in cloud computing," in *2016 Symposium on Colossal Data Analysis and Networking (CDAN)*. IEEE, 2016, pp. 1–4.

[4] Y. Xing and Y. Zhan, "Virtualization and cloud computing," in *Future Wireless Networks and Information Systems: Volume 1*. Springer, 2012, pp. 305–312.

[5] L. Malhotra, D. Agarwal, A. Jaiswal *et al.*, "Virtualization in cloud computing," *J. Inform. Tech. Softw. Eng*, vol. 4, no. 2, pp. 1–3, 2014.

[6] Y. Wang, G. Li, Z. Wang, Y. Kang, Y. Zhou, H. Zhang, F. Gao, J. Sun, L. Yang, P. Lee *et al.*, "Fast outage analysis of large-scale production clouds with service correlation mining," in *Proceedings of the 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 885–896.

[7] Z. Chen, Y. Kang, L. Li, X. Zhang, H. Zhang, H. Xu, Y. Zhou, L. Yang, J. Sun, Z. Xu *et al.*, "Towards intelligent incident management: why we need it and how we make it," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2020, pp. 1487–1497.

[8] J. Chen, S. Zhang, X. He, Q. Lin, H. Zhang, D. Hao, Y. Kang, F. Gao, Z. Xu, Y. Dang *et al.*, "How incidental are the incidents? characterizing and prioritizing incidents for large-scale online service systems," in *Proceedings of the 35th International Conference on Automated Software Engineering (ASE)*, 2020, pp. 373–384.

[9] S. Ghosh, M. Shetty, C. Bansal, and S. Nath, "How to fight production incidents? an empirical study on a large-scale cloud service," in *Proceedings of the 13th Symposium on Cloud Computing (SoCC)*, 2022, pp. 126–141.

[10] J. Liu, S. He, Z. Chen, L. Li, Y. Kang, X. Zhang, P. He, H. Zhang, Q. Lin, Z. Xu *et al.*, "Incident-aware duplicate ticket aggregation for cloud systems," *arXiv preprint arXiv:2302.09520*, 2023.

[11] M. Iliofotou, P. Pappu, M. Faloutsos, M. Mitzenmacher, S. Singh, and G. Varghese, "Network monitoring using traffic dispersion graphs (tdgs)," in *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement (SIGCOMM)*, 2007, pp. 315–320.

[12] Y. Jin, E. Sharafuddin, and Z.-L. Zhang, "Unveiling core network-wide communication patterns through application traffic activity graph decomposition," *ACM SIGMETRICS Performance Evaluation Review*, vol. 37, no. 1, pp. 49–60, 2009.

[13] S. Nagaraja, P. Mittal, C.-Y. Hong, M. Caesar, and N. Borisov, "Botgrep: Finding p2p bots with structured graph analysis." in *USENIX security symposium (USENIX security)*, vol. 10, 2010, pp. 95–110.

[14] K. Xu, F. Wang, and L. Gu, "Network-aware behavior clustering of internet end hosts," in *2011 proceedings ieee infocom (INFOCOM)*. IEEE, 2011, pp. 2078–2086.

[15] B. Arzani, S. Ciraci, S. Saroiu, A. Wolman, J. W. Stokes, G. Outhred, and L. Diwu, "Privateeye: Scalable and privacy-preserving compromise detection in the cloud." in *NSDI*, 2020, pp. 797–815.

[16] Z. Chen, J. Liu, Y. Su, H. Zhang, X. Ling, Y. Yang, and M. R. Lyu, "Adaptive performance anomaly detection for online service systems via pattern sketching," in *Proceedings of the 44th International Conference on Software Engineering (ICSE)*, 2022, pp. 61–72.

[17] G. Zhao, S. Hassan, Y. Zou, D. Truong, and T. Corbin, "Predicting performance anomalies in software systems at run-time," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 30, no. 3, pp. 1–33, 2021.

[18] T. Yang, J. Shen, Y. Su, X. Ling, Y. Yang, and M. R. Lyu, "Aid: efficient prediction of aggregated intensity of dependency in large-scale cloud systems," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 653–665.

[19] W. Pang, S. Panda, J. Amjad, C. Diot, and R. Govindan, "{CloudCluster}: Unearthing the functional structure of a cloud service," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2022, pp. 1213–1230.

[20] "Amazon cloudwatch documentation." [Online]. Available: https://docs.aws.amazon.com/cloudwatch/index.html

[21] "Overview of azure monitor alerts - azure monitor." [Online]. Available: https://docs.microsoft.com/en-us/azure/azure-monitor/alerts/alerts-overview

[22] "Cloud monitoring." [Online]. Available: https://cloud.google.com/monitoring/

[23] "Google cloud: Use vpc flow logs." [Online]. Available: https://cloud.google.com/vpc/docs/using-flow-logs

[24] "Logging ip traffic using vpc flow logs - amazon web services (aws)." [Online]. Available: https://docs.aws.amazon.com/vpc/latest/userguide/flow-logs.html

[25] "Microsoft azure: Flow logs for network security groups." [Online]. Available: https://learn.microsoft.com/en-us/azure/network-watcher/network-watcher-nsg-flow-logging-overview

[26] "Jaccard index - wikipedia." [Online]. Available: https://en.wikipedia.org/wiki/Jaccard_index

[27] M. Müller, "Dynamic time warping," *Information retrieval for music and motion*, pp. 69–84, 2007.

[28] T. T. Tanimoto, "Elementary mathematical theory of classification and prediction," 1958.

[29] J. Leskovec, A. Rajaraman, and J. D. Ullman, *Mining of massive data sets*. Cambridge university press (CAMB), 2020.

[30] A. Z. Broder, "On the resemblance and containment of documents," in *Proceedings. Compression and Complexity of SEQUENCES 1997 (SEQUENCES)*. IEEE, 1997, pp. 21–29.

[31] F. Nielsen and F. Nielsen, "Hierarchical clustering," *Introduction to HPC with MPI for Data Science*, pp. 195–211, 2016.

[32] D. Defays, "An efficient algorithm for a complete link method," *The Computer Journal*, vol. 20, no. 4, pp. 364–366, 1977.

[33] Z. Li, Y. Zhao, R. Liu, and D. Pei, "Robust and rapid clustering of kpis for large-scale anomaly detection," in *2018 IEEE/ACM 26th International Symposium on Quality of Service (IWQoS)*. IEEE, 2018, pp. 1–10.

[34] S. Zhang, D. Li, Z. Zhong, J. Zhu, M. Liang, J. Luo, Y. Sun, Y. Su, S. Xia, Z. Hu *et al.*, "Robust system instance clustering for large-scale web services," in *Proceedings of the ACM Web Conference 2022 (WWW)*, 2022, pp. 1785–1796.

[35] L. Li, X. Zhang, X. Zhao, H. Zhang, Y. Kang, P. Zhao, B. Qiao, S. He, P. Lee, J. Sun *et al.*, "Fighting the fog of war: Automated incident detection for cloud systems," in *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, 2021, pp. 131–146.

[36] H. Liu, S. Lu, M. Musuvathi, and S. Nath, "What bugs cause production cloud incidents?" in *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS)*, 2019, pp. 155–162.

[37] J. Gu, J. Wen, Z. Wang, P. Zhao, C. Luo, Y. Kang, Y. Zhou, L. Yang, J. Sun, Z. Xu *et al.*, "Efficient customer incident triage via linking with system incidents," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2020, pp. 1296–1307.

[38] A. Kane and N. Shiri, "Multivariate time series representation and similarity search using pca," in *Advances in Data Mining. Applications and Theoretical Aspects: 17th Industrial Conference, ICDM 2017*. Springer, 2017, pp. 122–136.

[39] J. Paparrizos and L. Gravano, "k-shape: Efficient and accurate clustering of time series," in *Proceedings of the 2015 ACM SIGMOD international conference on management of data (SIGMOD)*, 2015, pp. 1855–1870.

[40] P. Huang, C. Guo, L. Zhou, J. R. Lorch, Y. Dang, M. Chintalapati, and R. Yao, "Gray failure: The achilles' heel of cloud-scale systems," in *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS)*, 2017, pp. 150–155.

[41] H. S. Gunawi, M. Hao, R. O. Suminto, A. Laksono, A. D. Satria, J. Adityatama, and K. J. Eliazar, "Why does the cloud stop computing? lessons from hundreds of service outages," in *Proceedings of the Seventh ACM Symposium on Cloud Computing (SoCC)*, 2016, pp. 1–16.

[42] D. Cotroneo, L. De Simone, P. Liguori, R. Natella, and N. Bidokhti, "How bad can a bug get? an empirical analysis of software failures in the openstack cloud computing platform," in *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2019, pp. 200–211.

[43] N. Zhao, J. Chen, X. Peng, H. Wang, X. Wu, Y. Zhang, Z. Chen, X. Zheng, X. Nie, G. Wang *et al.*, "Understanding and handling alert storm for online service systems," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2020, pp. 162–171.

[44] Y. Chen, X. Yang, H. Dong, X. He, H. Zhang, Q. Lin, J. Chen, P. Zhao, Y. Kang, F. Gao *et al.*, "Identifying linked incidents in large-scale online service systems," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2020, pp. 304–314.

[45] Y. Chen, X. Yang, Q. Lin, H. Zhang, F. Gao, Z. Xu, Y. Dang, D. Zhang, H. Dong, Y. Xu *et al.*, "Outage prediction and diagnosis for cloud service systems," in *Proceedings of the 28th World Wide Web Conference (WWW)*, 2019, pp. 2659–2665.

[46] J. Chen, P. Wang, and W. Wang, "Online summarizing alerts through semantic and behavior information," in *Proceedings of the 44th International Conference on Software Engineering (ICSE)*, 2022, pp. 1646–1657.

[47] Z. Chen, J. Liu, Y. Su, H. Zhang, X. Wen, X. Ling, Y. Yang, and M. R. Lyu, "Graph-based incident aggregation for large-scale online service systems," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 430–442.