Incident-aware Duplicate Ticket Aggregation for Cloud Systems

Jinyang Liu*, Shilin He†, Zhuangbin Chen*, Liqun Li†, Yu Kang†, Xu Zhang†, Pinjia He‡, Hongyu Zhang§, Qingwei Lin†**, Zhangwei Xu¶, Saravan Rajmohan‖, Dongmei Zhang†, Michael R. Lyu*
*The Chinese University of Hong Kong, Hong Kong SAR, China, {jyliu, zbchen, lyu}@cse.cuhk.edu.hk
†Microsoft Research, Beijing, China, {shilin.he, liqli, kay, xuzhang2, qlin, dongmeiz}@microsoft.com
‡The Chinese University of Hong Kong, Shenzhen, China, hepinjia@cuhk.edu.cn
§Chongqing University, Chongqing, China, hyzhang@cqu.edu.cn
¶Microsoft Azure, Redmond, USA, zhangxu@microsoft.com
‖Microsoft 365, Redmond, USA, saravanakumarrajmohan@outlook.com

2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE) | 978-1-6654-5701-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICSE48619.2023.00193

***Abstract*—In cloud systems, incidents are potential threats to customer satisfaction and business revenue. When customers are affected by incidents, they often request customer support service (CSS) from the cloud provider by submitting a support ticket. Many tickets could be duplicate as they are reported in a distributed and uncoordinated manner. Thus, aggregating such duplicate tickets is essential for efficient ticket management. Previous studies mainly rely on tickets' textual similarity to detect duplication; however, duplicate tickets in a cloud system could carry semantically different descriptions due to the complex service dependency of the cloud system. To tackle this problem, we propose iPACK, an incident-aware method for aggregating duplicate tickets by fusing the failure information between the customer side (i.e., tickets) and the cloud side (i.e., incidents). We extensively evaluate iPACK on three datasets collected from the production environment of a large-scale cloud platform, Azure. The experimental results show that iPACK can precisely and comprehensively aggregate duplicate tickets, achieving an F1 score of 0.871∼0.935 and outperforming state-of-the-art methods by 12.4%∼31.2%.**

***Index Terms*—duplicate tickets, incidents, cloud systems, reliability**

I. Introduction

In the era of Cloud Computing, cloud platforms such as Amazon AWS, Microsoft Azure, and Google Cloud Platform serve millions of users worldwide. When customers encounter a technical problem with the platform; they often resort to cloud providers for help by submitting a *support ticket* (ticket for short), which consists of a textual issue description and some basic attributes (e.g., date and product name). From the cloud provider's perspective, once a ticket is received, it is essential to provide timely assistance to customers to avoid user dissatisfaction and financial loss [1][2].

In practice, *incidents* (i.e., unexpected service interruptions) are inevitable for large-scale cloud platforms [3][4]. Though much effort has been devoted to ensure the reliability of cloud systems [5][6][7], customers could still be impacted by incidents. For a large-scale cloud platform serving millions of customers, incidents could trigger a large number of tickets, among which many could be duplicate as the tickets are reported in a distributed and uncoordinated manner. To reduce the burden of support engineers, it is essential to *precisely and comprehensively aggregate the tickets, i.e., clustering the duplicate tickets caused by the same incident*. By doing this, the support team can resolve the tickets more efficiently.

To aggregate the tickets caused by the same incident, a common practice is to check if multiple tickets with similar symptom descriptions are reported within a short period. The intuition behind this is that customers using the same functionalities or services tend to encounter similar problems if they are caused by the same incident (e.g., service unavailability). Most existing studies on duplicate issue report detection measure the semantic similarity between two reports based on their textual descriptions, using natural language processing techniques such as word frequency [8][9][10], word embedding [11][12], topic modeling [13], and pretrained model [14]. Such semantic similarity-based approaches work well for traditional software systems (e.g., NetBeans [15], Eclipse and Firefox [16]). However, they are sub-optimal for aggregating duplicate tickets in cloud systems due to the large-scale and heterogeneous architecture of cloud systems [4][17][18]. The main reason is that customers of cloud systems could encounter distinct issues (with distinct symptoms) caused by the same incident. On the one hand, customers using the same service may experience different issues due to various usage scenarios. For example, when the control plane of the virtual machine (VM) service is problematic, the customers could complain about various problems related to VM creation, upgrade or deletion, depending on their particular scenarios. On the other hand, multiple services can be impacted by the same incident due to the notorious failure propagation problem [5][18][19] in cloud systems. For example, when an infrastructure-level service (e.g., a storage service) is interrupted, other services depending on it (e.g., VM and Web application) can be impacted too. As a result, customers using different services may observe different symptoms and submit tickets with dissimilar descriptions. Consequently, it is

**Qingwei Lin is the corresponding author.

1558-1225/23/$31.00 ©2023 IEEE
DOI 10.1109/ICSE48619.2023.00193

insufficient to tackle this problem by solely utilizing textual descriptions of tickets.

To address existing studies' limitations, we propose introducing cloud-side runtime information, i.e., *alerts*, to facilitate ticket aggregation in cloud systems. Modern cloud systems widely adopt monitors to continuously detect anomalies (unexpected behaviors) of cloud systems [20][21][22]. Once an anomaly is detected, an alert describing the anomaly will be fired to notify on-call engineers for inspection promptly. The services (and their internal components) are interdependent in cloud systems [17][18]; therefore, when an incident impacts multiple components or services, multiple alerts will be triggered within a short period [6][23], that is, these alerts are correlated with each other (i.e., **alert-alert relation**). According to our study in Azure, the correlated alerts caused by most (93%) incidents are fired within four hours. On the other hand, a particular issue of a component (e.g., problematic API for VM allocation) in cloud systems can reflect a particular customer-side issue (e.g., cannot create a VM). So, it is possible to find a *responsible alert* within the component that captures the issue resulting in the ticket (i.e., **ticket-alert relation**). In Azure, we find that for 92% of customer tickets; the alert system has already fired responsible alerts that cause these tickets before the tickets are submitted.

Motivated by these two kinds of relations, we propose to formulate the ticket aggregation problem in cloud systems as a two-stage linking problem, i.e., alert-alert linking and ticket-alert linking. Intuitively, if the same incident triggers multiple inter-linked alerts and these alerts are further linked to different tickets, then we consider these tickets should be aggregated (i.e., caused by the same incident). In doing this, it is possible to aggregate semantically different tickets via alert-alert links.

However, designing such a framework mainly faces two challenges originating from the large scale and complexity of cloud systems: First, alerts are massive and noisy. The main reason is that cloud systems consist of a large number of interdependent services. Each service adopts comprehensive monitors to capture any abnormal patterns to ensure its reliability [24]. These monitors could be sensitive. As a result, various alerts are continuously fired every second [5], so it is challenging to correctly identify and link alerts that are relevant to the ongoing incident. Second, features of both alerts and tickets have high cardinality, which means each of their features consists of too many unique values. When considering linking alerts and tickets, the number of feature combinations grows exponentially due to the high cardinality. Consequently, it is hard to identify effective feature combinations between them and conduct correct correlation.

In this paper, we propose iPACK to address these challenges. Specifically, iPACK mainly consists of three steps, i.e., *alert parsing*, *incident profiling* and *ticket-event correlation*. The first two steps address the first challenge, and the third step addresses the second challenge. In the *alert parsing* step, we preprocess (parse) alerts as more coarse-grained *events* to reduce redundant alerts. Next, in the *incident profiling* step, we propose GIP (graph-based incident profiling) to automatically filter noisy events and link events caused by the same incident. As a result, each incident is represented as an event graph by considering alert-alert relations. Afterward, in the *ticket-event correlation* step, we propose AIN (attentive interaction network) to correlate a ticket to a responsible event by considering ticket-alert relations. Finally, we aggregate these tickets that are linked to the events within the same event graph (i.e., incident), which are provided to CSS (customer support service) team to accelerate processing the tickets.

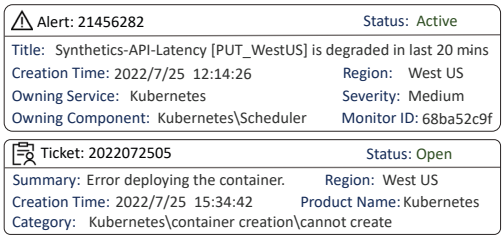


Fig. 1. An example of an alert and its resultant ticket.

This work makes the following major contributions:

- We are the first to propose to introduce cloud runtime information (i.e., alerts) to aggregate duplicate tickets. We propose iPACK to leverage the alert-alert relations and ticket-alert relations to achieve this goal.
- We evaluate iPACK on three datasets collected from the production environment of Azure. The evaluation results show that iPACK outperforms state-of-the-art methods by 12.4%∼31.2%, which confirm the effectiveness of iPACK. We also share our industrial experience of applying iPACK in a large-scale cloud platform, Azure.

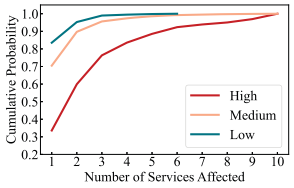
II. Background and Motivating Example

A. Background

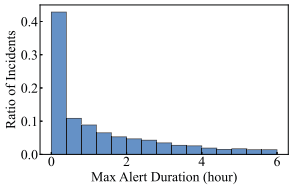
*a) **Alert**:* Alerts are fired by monitors that continuously detect anomalies in cloud systems, which automatically notify on-call engineers for investigation [17][25][23]. An alert has many attributes as presented in Fig. 1 (top), including *alert ID*, *title*, *creation time*, *region*, *owning service*, *owning component*, *severity*, *monitor ID*, etc. The *title* is generated by following a template pre-defined by engineers. The *severity* indicates how serious the issue is, which has three levels, i.e., low, medium and high. A service (*owning service*) consists of many components (*owning component*), where each component has its own functionality or feature.

*b) **Support Ticket**:* As presented in Fig. 1 (bottom), a support ticket usually contains attributes such as *ticket ID*, *creation time*, *summary*, *region*, *product name*, and *category*. The *summary* is free text written by customers in natural language. The *region* is where the customer's product is deployed. The *category* is a coarse-to-fine text description initially selected by the customer, which facilitates triaging a ticket to a proper support engineer. In addition, a ticket may also include a long detailed description (hidden in the figure).

Modern cloud platforms adopt similar schemes of alerts and support tickets described above. For example, CloudWatch of AWS [20], Alerting of GCP [22] and Azure Monitor [21] share



(a) The number of services impacted by incidents



(b) Distribution of max alert duration of incidents

Fig. 2. Statistics of alert and incident data in Azure.

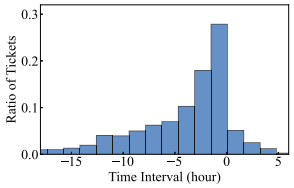


Fig. 3. Time interval between alerts and resultant tickets

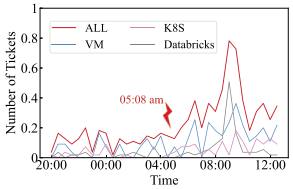


Fig. 4. Ticket number trend during an incident

similar alerting mechanisms, and their alerts carry similar attributes. Besides, their ticket management systems require similar attributes from customers as in Fig. 1, i.e., AWS Support [26], Google Support Hub [27], and Azure Support [27]. In this work, we only leverage the *common* features that all these popular cloud platforms own to ensure generalizability.

B. Alert-Alert Relation

The alert-alert relation denotes that two alerts could be correlated if they are caused by the same incident. The relation originates from the hierarchical structures of modern cloud systems that consist of inter-dependent components or services [24]. When an incident happens, multiple components or services could be impacted due to failure propagation [18][23], which will fire alerts within a short period associated with the same incident. During the diagnosis of an incident, in Azure, on-call engineers will manually mark these alerts and assess the severity of the incident according to the number of customers impacted. According to the diagnosis history in Azure from 2020/01/01 to 2022/06/01, as shown in Fig. 2(a), we found incidents with a higher severity tend to affect more services. Especially, 70% of high-severity incidents affect more than one service. We studied the resultant alerts of historical incidents. We calculated the max alert duration of the incidents (i.e., the time interval between the earliest and the latest alerts triggered by the incident). As shown in Fig. 2(b), we found that the max alert duration of 93% of incidents is within four hours. This serves as evidence to automatically identify the correlated alerts within an incident (in Section III-C).

C. Ticket-Alert Relation

The ticket-alert relation denotes that a ticket can correlate with a responsible alert inside the cloud systems. When a particular type of issue happens inside the cloud system (alerts are also fired), the customer could experience particular problems. Fig. 1 presents an example. If the API PUT (for container allocation) in the Kubernetes services is degraded, the customer can experience an error when deploying a container. In Azure, if a ticket is related to a cloud-side issue, the support engineers are required to annotate the responsible alert ID after diagnosis. Based on the annotated alert-ticket pairs collected from 2020/01/01 to 2022/06/01, we study the time interval between alert generation and ticket submission. Fig. 3 shows the results, where a negative time interval indicates that an alert is fired before the ticket is submitted. We found around 92% of tickets have responsible alerts fired before customers submit the tickets. This allows us to correlate responsible alerts for most tickets in runtime (in Section III-D).

For clarification, we summarize these important terminologies (i.e., alert, ticket, incident, alert-alert relation and ticket-alert relation) in Table I for reference.

TABLE I
TERMINOLOGY DEFINITION

Terminology	Definition
Alert	An alert is triggered when abnormal behavior of a component is detected.
Ticket	A request raised by a customer to ask the cloud vendor for help.
Incident	Unexpected interruptions affecting services' availability or performance, which usually trigger a series of alerts.
Alert-Alert Relation	Two alerts are correlated if they are caused by the same incident. (Section II-B)
Ticket-Alert Relation	A ticket is correlated with an alert if the former is caused by the latter. (Section II-C)

D. A Motivating Example

We present a real-world incident in July 2021 in Azure (the public link describing the incident is anonymized) and its resultant tickets as a motivating example. The impact of the incident started at 05:08 AM (UTC). It was caused by the availability loss of the DiskRP (disk resource provider) service that provides a control plane service for managed disks. Since its gateway queue was full, a large proportion of incoming requests were rejected. As a consequence, services relying on DiskRP experienced interruptions. On-call engineers' diagnosis confirmed that three services were impacted, i.e., virtual machine (VM), Databricks, and Kubernetes (K8S). Customers using these services were affected, which led to overwhelming tickets. As shown in Fig. 4, the ticket numbers of the services simultaneously increased right after the impact started, which implies the three services could be impacted by the same incident concurrently. In particular, the CSS team received around *four* times the number of tickets than usual within a short period and assigned *twice* the number of support engineers to handle these tickets. We list some samples of alerts and tickets related to this incident in Table II. These tickets ($t_1 \sim t_8$) carry dissimilar semantics due to different use scenarios and services for different customers. Therefore, it is hard to know that these tickets are actually caused by the same incident, rendering the difficulty for support engineers to group them and handle the burst of tickets efficiently.

TABLE II
ALERTS CAUSED BY THE SAME INCIDENT AND THE RESULTANT TICKETS (SOME FEATURES ARE OMITTED DUE TO SPACE LIMITATION.)

Service	Tickets		Alerts	
	Category	Summary	Component	Title
VM	VM/Scale Update	t_1: Virtual machine scale sets resize issue.	Resource Provider	a_1 : VMStart Failures exceed 300 times.
	VM/VM Start	t_2: Server did not start on time.		
Databricks	Databricks/Job Issue	t_3: Unable to open cluster of Databricks.	Control Plane	a_2 : Databricks cluster creation fails.
	Databricks/Cluster Launch	t_4: Unable to provision clusters.		
K8S	K8S/Cluster Update	t_5: Unable to autoscale.	Resource Scheduler	a_3 : The PUT operation success rate $<$80%.
	K8S/Cluster Update	t_6: Cannot upgrade node pool, stuck.		a_4 : CPU utilization exceeds 90%.

We propose to aggregate these tickets by simultaneously leveraging the aforementioned alert-alert relations and ticket-alert relations. We take Table II as an example to elaborate our intuition. First, we need to know what alerts are triggered by an incident, i.e., profiling the incident. In this example, we link the alerts $a_1-a_2-a_3$ via capturing the alert-alert relations (i.e., they are caused by the same incident). Second, we need to know what tickets are caused by these alerts, namely, linking $a_1-(t_1,t_2)$, $a_2-(t_3,t_4)$, and $a_3-(t_5,t_6)$. Finally, because the alerts $a_1 \sim a_3$ are linked as an incident and $t_1 \sim t_6$ are further linked to these alerts, we can aggregate $t_1 \sim t_6$ as the same cluster even though they possess dissimilar semantics.

Challenges. To achieve this, iPACK should address the following two challenges originated from the large scale and complicated architecture of cloud systems [5][18][24].

Challenge 1: Massive and noisy alerts. Cloud systems could contain thousands of interdependent services. These services are closely monitored from various aspects to capture any unexpected behaviors. For example, there could be hundreds, even thousands of high-severity alerts reported in Azure per day. Some alerts are **regular alerts** that are reported frequently (due to sensitive monitoring rules) and periodically (due to periodical monitoring). These regular alerts are generally not related to a particular cloud incident and only report usual system runtime status such as CPU/memory usage rate (e.g., a_4 in Table II). In contrast, **indicative alerts** are caused by an actual problem of cloud systems. For example, the alerts $a_1 \sim a_3$ in Table II are indicative alerts. It is challenging to identify the indicative alerts and correctly link them among massive and noisy alerts.

Challenge 2: High feature cardinality. High feature cardinality refers to a situation where a feature has a large number of unique values. For example, the feature *category* of a ticket has more than 3,000 options, and the features *component* and *monitor ID* of alerts have more than 2,000 and 10,000 options, respectively. Using traditional one-hot encoding [28] methods to process these features would lead to a high-dimensional feature space, resulting in the curse of dimensionality [29]. Additionally, linking alerts to tickets requires the consideration of various combinations of features between them. However, due to the high feature cardinality, the number of possible combinations grows exponentially, making it difficult to identify the most effective combinations that accurately reflect the correlation between alerts and tickets. This constitutes a significant challenge in our work.

III. METHODOLOGY

A. Overview of iPACK

The goal of iPACK is to aggregate duplicate tickets that are caused by the same cloud incident among all tickets. Due to the large scale and heterogeneous architecture [4][17][18] of cloud systems, it is insufficient to solely consider the textual similarity of tickets to achieve this goal. To address this problem, we introduce cloud run-time information (i.e., alerts) and formulate it as a two-stage linking problem. Intuitively, iPACK first finds links between alerts by leveraging alert-alert relations. These inter-linked alerts constitute a graph to represent an incident. Then iPACK identifies the tickets that are caused by these alerts according to ticket-alert relations. The tickets linked to the alerts within the same graph (i.e., incident) are aggregated. Thus, we can aggregate the tickets with dissimilar semantics via the bridge of alert-alert links.

As shown in Fig. 5, iPACK consists of three steps: *alert parsing*, *incident profiling* and *ticket-event correlation*. In the *alert parsing* step, we parse alerts as more coarse-grained *events* to reduce redundant alerts. Next, in the *incident profiling step*, we propose a graph-based incident profiling (GIP) method to remove the regular events (i.e., parsed regular alerts) and link correlated indicative events. Then, in the *ticket-event correlation*, we propose an attentive interaction network (AIN) to correlate a ticket to an event. Finally, if two tickets are correlated to the events within the same event graph (i.e., the same incident), we aggregate the tickets as the same cluster. The results of the ticket aggregation are presented to the CSS (Customer Support Services) team to streamline the ticket processing process and improve efficiency. This allows support engineers to send out batch notifications to potentially affected customers and provide quick guidance for service recovery. Additionally, the results can aid on-call engineers in conducting impact assessments, including identifying affected services and determining the extent of customer impact caused by the incident (e.g., number of affected customers).

B. Alert Parsing

The title of an alert is generated following an engineer-specified template. Monitors may be triggered multiple times during an incident causing massive redundant alerts. To reduce the volume of alerts and avoid redundancy, we parse each alert to its corresponding template and aggregate the alerts sharing the same template as an **event**. Take a_1 in Table II as an example; multiple similar alerts can fire concurrently such as

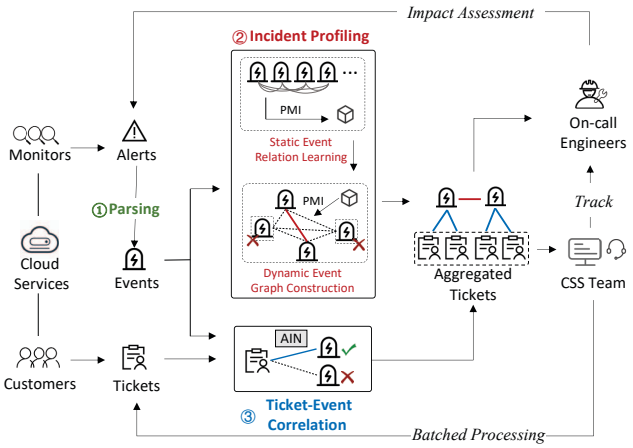


Fig. 5. The overall framework of iPACK.

"VMStart Failures exceed 100/150/200/250 times", which are aggregated as "VMStart Failures exceed $*$ times".

We formulate this problem as the well-studied log parsing problem [30] following [18]. We propose to customize a widely-adopted log parsing algorithm, Drain [31] to parse the alerts into templates (events). Drain works by extracting the common parts of alert titles from each group of alerts, where the group is determined by calculating the overlap of words. To enhance Drain in our scenario, we observe that if two alerts are reported by different monitors or belong to different components, the two alerts must have distinct templates. Therefore, we first divide all alerts into different partitions according to both *monitor ID* and *owning component*. We then apply Drain in each partition to extract the templates. In this way, we can reduce the noises in each partition and also accelerate the processing by parallel computing. Finally, each alert is parsed as an event, which introduces two features, i.e., *event template* and *event ID* (a hash value of its template). Within a fixed time window (Section III-C), for events sharing the same template, we reserve the latest event and discard the rest of the events to reduce its volume. The following steps are applied to events instead of raw alerts.

C. Incident Profiling

The goal of this step is to represent an incident via *linking the correlated events that are caused by the same incidents*. In doing this, the linked events can then be used to bridge semantically different tickets in the next step (Section III-D).

To learn relations between events, some existing solutions leverage manual annotations [24][32], which are not practical because such labels are hard to obtain and usually insufficient in real-world practice. While there are unsupervised solutions [6][19], they require prior knowledge (e.g., the precise topology of cloud services) to estimate alert relations. However, such prior knowledge is usually inaccurate and requires extensive efforts to collect, update and validate [17][19][24].

We propose an unsupervised approach, i.e., Graph-based Incident Profiling (GIP), which does not rely on prior knowledge. The input is a series of events within a time window, and the output is one or multiple graphs of the events. Each graph profiles an incident containing indicative events related to the incident. GIP has a *static event relation learning* step and a *dynamic event graph construction* step. Intuitively, if two events are correlated, these events are more likely to be triggered within a short period frequently in the history [19][24]. We model such frequent patterns in the first *static event relation learning* step. Then, in the *dynamic event graph construction* step, we dynamically link the events possessing the learned frequent patterns and remove regular events in the runtime.

1) Static Event Relation Learning: In this step, we assign a static score to each event pair weighing how likely they co-occur in history. To this end, we first collect a series of historical events in chronological order. Then we apply a *four-hour-long* sliding time window on these events with a step size of one hour. We adopt *four* hours as the window length because it can cover most alerts within an incident according to our study in Section II-B. The one-hour step size allows us to introduce enough new events for learning the static event relations and avoid separating co-occurred events into two different windows. Each window w_i contain multiple events, i.e., $w_i = [e_1, e_2, e_3, ...]$. If two events appear in the same window, we count it as a co-occurrence. Based on these windows, we compute the point-wise mutual information (PMI) score [33] for each event pair, which is a popular metric to measure co-occurrence associations [34][35]. Formally, the PMI value for the event pair (e_i, e_j) is :

$$PMI(e_i, e_j) = log\frac{p(e_i, e_j)}{p(e_i)p(e_j)}, \tag{1}$$

where $p(e_i, e_j) = \frac{C(e_i, e_j)}{M}$, $p(e_i) = \frac{C(e_i)}{M}$. $C(e_i, e_j)$ denotes the number of windows that contain both e_i and e_j, and $C(e_i)$ is the number of windows that contain e_i. M is the total number of windows. A higher PMI value indicates two events are more likely to co-occur in history, and a positive PMI value indicates they are more likely to co-occur than appear individually. We use $d(e_i, e_j)$ to denote the pre-computed PMI value for the event pair (e_i, e_j).

2) Dynamic Event Graph Construction: We then dynamically construct event graphs in the runtime by utilizing the learned static PMI values. The input to this step is the events collected within the latest four-hour-long time window. The output is one or more event graphs, each of which contains correlated events caused by the incident.

Intuitively, we aim to link the events with high PMI values because they are possibly caused by the same ongoing incident in the runtime, considering they frequently co-occur in history. However, regular (noisy) events tend to co-occur with various types of events because they frequently appear regardless of whether there is an incident. In contrast, indicative events only frequently co-occur with only a small portion of events. Based on the difference between regular events and indicative events, we propose a novel algorithm to prune the regular events automatically, and the remaining indicative events are correlated. The pseudocode of the algorithm is shown in Algorithm 1. First, we link every pair of events with positive

Algorithm 1: Dynamic Event Graph Construction

Input: Pre-computed PMI values in d, a window of latest events $w_j = [e_1, e_2, e_3, ...]$, hyper-parameter $\mu \in [0,1]$
Output: $g_o = \{g_1, g_2, ...\}$
Init: g $\leftarrow$ Empty undirected graph; r $\leftarrow$ Empty list

1 **for** $i \leftarrow 1$ **to** l **do**
2 **for** $j \leftarrow i$ **to** l **do**
3 **if** $d(e_i, e_j) > 0$ **then**
4 g.AddWeightedEdge((e_i, e_j), weight=$d(e_i, e_j)$)
5 **end**
6 **end**
7 **end**
8 **for** *each node* $e_i \in g$ **do**
9 $\mathcal{W}$ = GetWeightsOfOutEdges(e_i)
10 AscendingSort($\mathcal{W}$)
11 γ = SearchKneePoint($\mathcal{W}$) // Kneedle algorithm
12 **if** $\gamma < \mu$ **then**
13 g.RemoveNode(e_i)
14 **end**
15 **end**
16 $g_o \leftarrow$ GetSubGraphs(g)

PMI values constituting a single initial event graph g with the PMI values as weights of edges (line $1 \sim 7$). Then, for each node, we calculate a knee point (i.e., γ in Algorithm 1) based on the PMI values of all its out edges, i.e., $\mathcal{W}$ (line $9 \sim 11$). Specifically, we adopt the Kneedle algorithm [36] to calculate γ. A small γ for a node denotes that most PMI values of its linked neighbors are large, namely, the node frequently co-occurs with many neighbors (events). This implies that the node is more likely to be a regular event. Therefore, we remove the node if its γ is less than a threshold μ (line $12 \sim 14$). As revealed by previous studies [6][37], regular events make up a large portion of all events. Therefore, we empirically set $\mu = 0.8$ to remove most events aggressively, which turns out to be effective in our scenario (Section IV-C3). Finally, we extract subgraphs (i.e., connected component [38]) from the the pruned graph g (line 16).

D. Ticket-Event Correlation

After profiling incidents as several event graphs (i.e., event-event linking), we correlate each ticket to the event that captures the internal cloud issue resulting in the ticket (i.e., event-ticket linking). If two tickets are correlated to inter-linked events (i.e., they are caused by the same incident), we can then aggregate them as the same cluster.

We mainly address the challenge caused by the high cardinality of features of tickets and events (Section II-D). Inspired by factorization machine [39] in the field of recommendation systems, we propose an attentive interaction network (AIN), which decomposes feature combinations as Hadamard products of low-dimension feature embeddings. In this way, we bypass directly encoding the exponentially-growing feature combinations with high-dimension feature vectors. The input of AIN is a ticket-event pair and the output is a probability representing how likely the input pair is correlated. Fig. 6 shows the overall framework of AIN composed of three layers, i.e., *embedding layer*, *attentive interaction layer*, and *prediction layer*, which are elaborated as follows.

Embedding Layer. The embedding layer represents all features (f_i for a ticket feature and $\hat{f}_i$ for an event feature) as trainable vectors (i.e., embeddings) denoted as $\mathbf{v_i} \in \mathbb{R}^k$, where k is a user-defined hyper-parameter. For *summary* of tickets and *event template* of events (denoted as f_1 and $\hat{f}_1$ in Fig. 6), we resort to the power of pretrained model BERT (Bidirectional Encoder Representations from Transformers) [40] to embed their semantics as vectors. We exclude the detailed ticket description since it potentially introduces noises, and the summary already provides the essential part [11][14][41]. The remaining features are initialized as random vectors.

Attentive Interaction Layer. After each feature is associated with an embedding vector, the attentive interaction layer models the feature combination of two features as the Hadamard product (i.e., element-wise product denoted as $\odot$) of their corresponding embedding vectors. For $\mathbf{u} = \mathbf{x} \odot \mathbf{y}$ we have $u_i = x_i y_i$. The attentive interaction layer models combinations of features across a ticket and an event, formally,

$$\mathbf{z} = \sum_{i}^{n} \sum_{j}^{m} a_{ij} (\mathbf{v}_i \odot \mathbf{v}_j), \tag{2}$$

where n and m are the numbers of ticket and event features, respectively. To identify the effective feature combinations for different ticket-event pairs, AIN computes an importance score a_{ij} for each combination result $v_i \odot v_j$ in Equation (2). Afterwards, these feature combinations are summarized as a single representation $\mathbf{z} \in \mathbb{R}^k$ by computing their weighted average. The importance weight a_{ij} is calculated as follows:

$$\hat{a}_{ij} = \mathbf{h}^T \phi(\mathbf{W}(\mathbf{v}_i \odot \mathbf{v}_j) + \mathbf{b}), \tag{3}$$

$$a_{ij} = \frac{e^{\hat{a}_{ij}}}{\sum_i^n \sum_j^m e^{\hat{a}_{ij}}}, \tag{4}$$

Equation (3) denotes a fully-connected (FC) neural network that takes the combination of two features as input and outputs their (unnormalized) importance weight. where $\phi(x) = max(0, x)$ is the ReLU activation function. $\mathbf{h}^T \in \mathbb{R}^r$, $\mathbf{W} \in \mathbb{R}^{(r \times k)}$ and $\mathbf{b} \in \mathbb{R}^r$ are trainable parameters. r is a hyper-parameter that denotes the size of the hidden layer. Equation (4) normalizes the importance weights to $[0, 1]$. The importance weights control how much each feature combination contributes for prediction. For example, in Equation (2), for a_{ij} close to 1, its corresponding feature combination will dominate the summarized vector $\mathbf{z}$. This means that the prediction mostly depends on the feature combination of $\mathbf{v_i}$ and $\mathbf{v_j}$. In addition, the weights are automatically learned by the FC in Equation 3, we actually force AIN to select the effective feature combinations when learning from the data.

Prediction Layer. We formulate ticket-event correlation as a binary classification problem. Particularly, to calculate the correlation probability p, an FC neural network is applied on $\mathbf{z}$, i.e., $p = \sigma(\mathbf{w}_o^T \mathbf{z} + b_o)$, where $\mathbf{w}_o \in \mathbb{R}^k$ and $b_o \in \mathbb{R}$ are trainable parameters, and $\sigma(x) = \frac{1}{1+e^{-x}}$ is the Sigmoid function producing a probability within the range of $[0, 1]$. To update all trainable parameters, we utilize the popular Adam

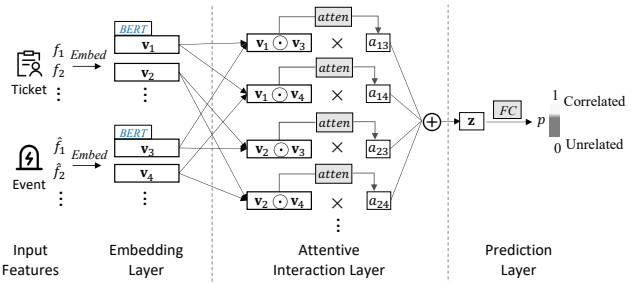


Fig. 6. The overall framework of AIN.

optimizer [42] to minimize the following binary cross-entropy loss $\mathcal{L}_{BCE}$ via fitting training data with N ticket-event pairs.

$$\mathcal{L}_{BCE} = -\sum_{i}^{N} \big(y_i log(p_i) + (1-y_i) log(1-p_i)\big), \qquad (5)$$

where $y_i = 1$ for positive (i.e., correlated) ticket-event pairs and $y_i = 0$ for negative (i.e., unrelated) pairs. The positive samples are collected by extracting the responsible alert ID of a ticket from its resolution text written by support engineers (Section II-C). So, such data is gradually accumulated during the daily work routine of support engineers, which does not incur additional manual effort for data labeling. We then randomly sample the same number of negative pairs. The features used are *event template*, *event ID*, *severity*, *monitor ID*, *owning service*, *owning component* for events, and *product name*, *category*, *summary* for tickets.

E. Deployment

iPACK consists of offline parts (pre-computed) and online parts (serving online continuously) for deployment in the could systems. The offline parts include alert parsing, static event relations learning and AIN training. The online parts conduct alert parsing, dynamic event graph construction, and ticket-event correlation utilizing the trained AIN. The details are as follows.

1) Offline Parts: Intuitively, the offline parts leverage the historical data to prepare intermediate data (e.g., PMI values) or model (e.g., AIN) for online use. Specifically, iPACK parse all collected alerts to events (Section III-B). Then, static event relations learning is conducted (Section III-C1), which computes PMI values for all event pairs. The PMI values are then stored in a Redis database for reference. After that, AIN is trained using historical ticket-event pairs. Azure continuously collects the alert and ticket data; in order to capture the latest system update (e.g., new alerts), the offline parts are executed periodically (e.g., once every month).

2) Online Parts: In the online deployment, iPACK is periodically executed (e.g., every five minutes) and pushes its latest analysis results to the CSS team. Support engineers can also manually trigger iPACK when needed (e.g., a large volume of tickets are received). Considering cloud services and customers are physically isolated in different regions, iPACK is applied separately in different regions. Once executed, iPACK collects the latest alerts and tickets within the latest four-hour-long time window to analyze. We can reduce the great volume of ticket-event pairs by filtering with region and time. The tickets and alerts in the same time window and region constitute a *chunk*.

In each chunk, after parsing alerts as events, GIP is applied to link events as event graphs (i.e., incidents). Then, we apply AIN to link each ticket to one of the events. For each ticket, AIN recommends a list of events ranked by the associated correlation probabilities. Note that we exclude the tickets whose largest probability in the ranked list is smaller than a confidence threshold $\theta = 0.8$, because they are more likely caused by a customer-side issue (e.g., incorrect configurations). Next, tickets that are correlated to the events within the same event graph are aggregated as a cluster. Based on the aggregation results, on the one hand, on-call engineers can conduct impact assessment (i.e., how many customers are impacted) for an incident; on the other hand, the CSS team can avoid duplicate manual inspection and make batched communication to customers. (e.g., provide the latest mitigation progress of the internal incident).

IV. Experiments

We answer the following research questions (RQs) to evaluate the performance of iPACK:

- **RQ1**: How effective is iPACK in aggregating duplicate tickets caused by the same incident?
- **RQ2**: How effective is AIN in correlating a ticket to the responsible event?
- **RQ3**: How does graph-based incident profiling (GIP) impact the effectiveness of iPACK?

A. Experimental Setting

1) Dataset: We collect the datasets from the production environment of Azure from 2020/01/01 to 2022/06/01. To evaluate the generality of iPACK, we collect three datasets from different physically isolated regions (i.e., $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$), which cover 81 services serving different numbers of customers. Each dataset is collected from tens of services and includes hundreds of incidents and hundreds of thousands of alerts. For each incident, the datasets contain tens of to hundreds of resulting tickets. Note that we hide the specific figures of the dataset statistics due to the confidential policy of Azure. We use the data before 2022/01/01 to compute PMI values (Section III-C) and train AIN (Section III-D). The data after the date is used for evaluation.

2) Comparative solutions: Recent studies have been working on user feedback analysis such as duplicate bug report detection [41][43][44][13][45] and emerging issue detection [46][10][47]. We select the following state-of-the-art approaches as our comparative solutions:

Categorization. We aggregate tickets by referring to their feature *category* (Section II-A), i.e., if two tickets share the same category, then they are aggregated into the same cluster.

iFeedback. iFeedback is proposed and adopted by WeChat in their production environment [10], which targets aggregating similar user feedback by identifying frequent word

combinations (and groups of combinations). For example, if the word combination of "pay" and "fail" bursts, an issue may happen to the payment feature of the product.

LWE. LWE [13] is a method integrating Latent Dirichlet Allocation (LDA) and word embeddings to leverage the advantages of both techniques. LWE first utilizes LDA to represent all tickets and roughly identify candidates of duplicated tickets. Then, the candidates are represented using word embeddings to conduct more fine-grained clustering.

BERT. BERT [40] is a popular pretraining model in natural language processing and has shown its power in capturing the semantics of user feedback in recent studies [14][48][49]. Because these studies do not directly aggregate user feedback, in this work, we adopt BERT to first represent the tickets as dense vectors, based on which we use agglomerative hierarchical clustering [50] to aggregate tickets.

LinkCM. LinkCM [51] is proposed to facilitate the triage of a customer-reported alert by matching it with an alert of cloud systems. LinkCM learns the correlation by purely fusing the titles between the report and alert via a decomposable attention mechanism and transfer learning. In our scenario, if two tickets are correlated to the same event by LinkCM, they are grouped together. LinkCM can also link a ticket to an event as AIN does, so we combine GIP with LinkCM (i.e., **LinkCM w/ GIP**) as a strong baseline for comparison.

3) Implementation Details: We have implemented iPACK with approximately 3000 lines of Python code and packaged it as a serverless function [17] for ease of use in Azure. The iPACK system is deployed on a CentOS Linux server with 60GB of RAM and an Intel(R) Core(TM) i7-5930K CPU @ 3.50GHz. The AIN component of iPACK is trained and tested with the GPU acceleration of an NVIDIA GeForce GTX TITAN X. We have set the default hyper-parameters of the AIN as k=128 and r=256, and the model is trained until its training loss stops decreasing for ten continuous epochs, using an early stopping approach. As for the comparative solutions, as they are not open-sourced, we have followed the implementation in their respective papers and leveraged well-established libraries to ensure accuracy. For example, we have used AllenNLP [52] for LinkCM, scikit-learn [53] and gensim [54] for LWE, and HuggingFace [55] for BERT.

B. Evaluation Metrics

Metrics for evaluating ticket aggregation (RQ1 and RQ3). Given a sequence of tickets, our approach assigns a unique cluster ID, denoted as "incident-number" to tickets that are caused by the same incident. Tickets that are not related to a cloud-side issue are marked with the cluster ID "non-incident". To evaluate the accuracy of our ticket aggregation, we use the widely accepted Rand Index [56], [57], [58] for pair-wise comparison in clustering. We conduct pair-wise comparisons between the ground-truth cluster ID and the predicted cluster ID for all tickets. The results are used to calculate the following metrics: True Positives (TP), which are pairs of duplicate tickets correctly predicted to have the same cluster label; True Negatives (TN), which are pairs of non-duplicate tickets correctly predicted to have different cluster labels; False Positives (FP), which are pairs of non-duplicate tickets wrongly predicted to have the same cluster label; and False Negatives (FN), which are pairs of duplicate tickets wrongly predicted to have different cluster labels. Based on the results, we use the following metrics to evaluate the aggregation results: $precision = \frac{TP}{TP+FP}$, $recall = \frac{TP}{TP+FN}$, and $F1\ score = 2 \cdot \frac{precision \cdot recall}{precision + recall}$.

Metrics for evaluating ticket-event correlation (RQ2). The correlation of tickets with an event, referred to as AIN in Section III-D, is a crucial component of iPACK. This component generates a ranked list of potential responsible events for a given ticket based on the probability scores (as determined by AIN's output) in descending order. To assess the accuracy of this step, we use the metric Acc@K (accuracy@K). For each ticket, if the actual ground-truth event appears within the top-K positions of the list, we consider the ticket to be a "hit". The Acc@K metric is calculated as the ratio of the number of hit tickets to the total number of tickets, represented as $Acc@K = \frac{\#\ of\ hit\ tickets}{\#\ of\ all\ tickets}$. In our evaluation, we consider three values of K (i.e., 1, 2, and 3) and also compute the average of these three metrics to provide a comprehensive assessment.

C. Experimental Details

*1) **RQ1** The Effectiveness of iPACK:* In this RQ, we aim to evaluate how accurately iPACK can aggregate the duplicate tickets by comparing it with all comparative solutions (Section IV-A2). The evaluation is conducted using datasets $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$ and the results are reported in terms of precision, recall, and F1 score. Precision reflects the degree of correctness in the clustering results, while recall represents the completeness of the results. The F1 score is a balance between precision and recall and provides a comprehensive measure of the overall performance of the approach. The results are presented in Table III. The highest F1 score is emphasized in **bold**, and the second-best score is underlined.

We can make the following observations: (1) iPACK achieves the best F1 score across all three datasets, i.e., 0.935, 0.871, and 0.894, outperforming the second-best methods by 31.2%, 12.4% and 18.4% in dataset $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$, respectively. (2) Categorization can achieve the highest precision (0.930∼0.943) although its recall is considerably low (0.205∼0.373). The reason is that the ticket feature *category* is defined in a fine-grained manner by support engineers in Azure. Therefore, it tends to aggressively split the complete set of duplicate tickets into many small groups, leading to a low recall score. However, tickets in each such small group share precisely similar semantics as evidenced by the high precision. (3) iFeedback, LWE, and BERT show lower precision but higher recall than Categorization. The reason is that these methods can capture more coarse-grained semantic similarity between tickets. Consequently, they can generate larger clusters (higher recall) but introduce additional noises (lower precision) (4) LinkCM can achieve a higher precision

TABLE III
EFFECTIVENESS OF AGGREGATING DUPLICATE TICKETS CAUSED BY THE SAME CLOUD INCIDENT.

Methods	Dataset $\mathcal{A}$			Dataset $\mathcal{B}$			Dataset $\mathcal{C}$		
	Precision	Recall	**F1 score**	Precision	Recall	**F1 score**	Precision	Recall	**F1 score**
Categorization	0.930	0.205	0.336	0.943	0.373	0.535	0.925	0.207	0.338
iFeedback	0.901	0.590	0.713	0.876	0.473	0.614	0.886	0.626	0.733
LWE	0.862	0.453	0.594	0.824	0.515	0.634	0.861	0.672	0.755
BERT	0.884	0.587	0.705	0.854	0.710	0.775	0.843	0.629	0.720
LinkCM	0.931	0.507	0.657	0.892	0.538	0.671	0.901	0.628	0.740
LinkCM w/ GIP	0.900	0.685	0.778	0.886	0.756	0.816	0.899	0.809	0.852
iPACK	0.912	0.960	**0.935**	0.882	0.861	**0.871**	0.899	0.888	**0.894**

among all baseline methods except Categorization. Moreover, after combining with GIP, LinkCM w/ GIP can increase its recall because more tickets are aggregated together through event-event linking. However, it still under-performs iPACK in terms of the overall F1 score because LinkCM cannot correlate a ticket to an event as accurately as iPACK does (will show in RQ2). For instance, LinkCM may associate a cluster of similar tickets with the wrong event. Therefore, even though related events are linked together, similar tickets are separated into different clusters, resulting in high precision but low recall.

Answer to RQ1. iPACK achieves the best F1 score among all state-of-the-art baselines across three datasets collected from different regions. iPACK slightly sacrifices precision compared with the Categorization method but achieves the highest F1 score 0.871∼0.935, outperforming state-of-the-art methods by 12.4%∼31.2%.

*2) **RQ2** The Effectiveness of ticket-event correlation:* In this RQ, the focus is on evaluating the accuracy of the ticket-event correlation step of iPACK, i.e., the proposed attentive interaction Network (AIN). The performance of AIN is compared with LinkCM [51] and four popular machine learning algorithms: LR (logistic regression), SVM (support vector machine), RF (random forest), and LightGBM (light gradient boosting machine). Additionally, the contribution of the attentive feature interaction component to AIN is studied.

To ensure a fair comparison, categorical features are represented as one-hot vectors, which are then concatenated with the representation of textual features extracted using BERT. This allows for a consistent input feature representation for all models compared. A variant of AIN is also developed by removing its attentive feature interaction component (referred to as "AIN w/o atten." in Table IV). This variant instead concatenates all feature embeddings into a single feature vector as the input for the prediction layer, as illustrated in Fig. 6. For clarity, this experiment is conducted using all pairs of ticket-event data from datasets $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$. We compare AIN with the baselines and its variant in terms of Acc@1, Acc@2, Acc@3 and the average of these metrics.

We can make the following observations in the results shown in Table IV: (1) The proposed AIN model outperforms all baseline models in terms of all four evaluation metrics. Notably, AIN achieves the highest Acc@1 score of 0.817, indicating its superior ability in accurately linking tickets to events and facilitating more effective ticket aggregation. (2) The introduction of the attentive feature interaction component results in significant improvements in AIN's performance, with a 21.4% increase in Acc@1 and a 17.8% increase in the average accuracy. This demonstrates that the component plays a crucial role in identifying effective feature combinations for accurate ticket-event linking. (3) Interestingly, AIN w/o atten. underperforms LinkCM and achieves similar performance as LightGBM. The reason is that AIN w/o atten. adopts simple concatenation of feature embedding, which fails to capture effective feature combinations. (4) LinkCM can outperform other baseline methods since its decomposable attention mechanism is able to capture the semantic matching between tickets and events. On the other hand, the relatively low Acc@1 scores of LR, SVM, RF and LightGBM may be due to the sparsity and high dimensionality of the input features. However, RF and LightGBM exhibit improved accuracy over LR and SVM, as they alleviate these problems through feature selection.

TABLE IV
EFFECTIVENESS OF CORRELATING A TICKET TO AN EVENT.

Models	Acc@1	Acc@2	Acc@3	Average
LR	0.519	0.657	0.733	0.636
SVM	0.332	0.409	0.493	0.411
RF	0.563	0.684	0.761	0.669
LightGBM	0.658	0.723	0.832	0.712
LinkCM	0.743	0.769	0.882	0.798
AIN w/o atten.	0.673	0.762	0.824	0.753
AIN	**0.817**	**0.907**	**0.936**	**0.887**
Δ(%)	+21.4%	+19.0%	+13.6%	+17.8%

Answer to RQ2. AIN outperforms all other baseline methods by a large margin in correlating a ticket to the event that causes it. The proposed attentive feature combination is the key to achieve the performance, which improves the average accuracy of AIN by 17.8%.

*3) **RQ3** The impact of graph-based incident profiling (GIP) of iPACK:* We propose GIP to reduce regular events (noisy

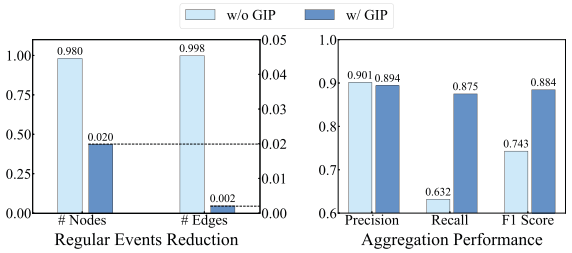


Fig. 7. The Effectiveness of Graph-based Profiling (GIP)

events) and link correlated indicative events to profile an incident, which bridges the tickets linked to the events even though they are semantically different. We evaluate its impact on iPACK using the union of all three datasets as in RQ2. We conduct the evaluation from the following two aspects:

(1) *The ratio of events reduced.* GIP builds a fully-connected event graph (link every two events with positive PMI values) and then prunes this graph via Algorithm 1. We measure the effectiveness of GIP with the ratio of nodes and edges that are pruned (reduced). Fig. 7 (left) presents the ratio of nodes and edges in the event graph without or with GIP (we normalize the ratio for better presentation). We can observe that only 2% of nodes and 0.2% of edges remain after using GIP, which shows GIP can reduce the large volume of events effectively.

(2) *The impact on the overall performance in aggregating duplicate tickets.* Though GIP can reduce the number of events, we aim to further evaluate whether it can accurately remove the regular events and link the correlated events as expected. To achieve this, we compare the ticket aggregation performance of iPACK with or without GIP. After removing GIP, we regard those tickets linked to the same event by AIN as belonging to the same cluster. The results are shown in Fig. 7 (right). We can observe that after applying GIP, its precision drops slightly, but the recall is largely improved. As a result, the overall F1 score is improved by 18.9%, from 0.743 to 0.884. This indicates that only a small portion of events are not correctly linked; however, more duplicate tickets are accurately aggregated via event-event linking.

Answer to RQ3. GIP can greatly boost the overall performance of iPACK. On the one hand, GIP reserves only 2% nodes and 0.2% edges in the pruned event graph. On the other hand, GIP accurately reserves and links the indicative events and improves the F1 score from 0.743 to 0.884.

V. Industrial Experience

In this section, we share our industrial experience by presenting a success case and a failure case from the real-world deployment of iPACK in Azure.

A. A success case

In September 2021, a datacenter maintenance activity resulted in the accidental shutdown of a water tower pump, which is a critical component of cooling systems. To prevent overheating and potential damage to users' data, the maintenance personnel had to shut down the downstream storage

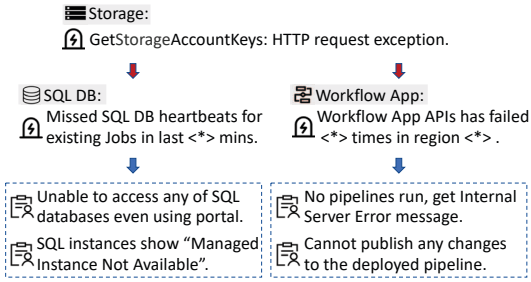


Fig. 8. A success case of iPACK in Azure

hardware. This caused a storage service disruption, leading to cascading impacts on several dependent services such as the SQL DB and Workflow App, and triggering alerts.

The CSS team received a substantial number of tickets describing a wide range of issues in response to these events. To assist with the situation, iPACK continuously collected and analyzed the generated alerts and tickets. The partial output of iPACK's analysis is presented in Fig. 8. iPACK successfully linked the storage alert with corresponding alerts from SQL DB and Workflow App, as demonstrated by the **red** arrows in Fig. 8. Additionally, the tickets caused by these events were linked to their respective root cause events, as depicted by the **blue** arrows. This allowed the tickets to be aggregated, despite their semantic differences, and the results were pushed to the support engineers. With the information provided by iPACK, support engineers were able to initiate batch communications with potentially impacted customers and avoid duplicative manual inspections. Throughout the resolution process, the customers were continuously informed of the mitigation progress of the incident.

B. A failure case

iPACK could sometimes fail when it cannot find responsible alerts in the cloud systems for a ticket. In August 2021, the CSS team received multiple tickets complaining of 503 (service unavailable) errors when the customers were using Web Services. Though the tickets were suspected to be caused by an internal issue due to their similar symptoms, iPACK did not correlate them with any alert. Only around five hours after the first ticket had been received, a related alert was fired and correlated by iPACK. According to the after-the-fact analysis of on-call engineers, the root cause of this incident turned out to be bad configurations of a Canary (gray) release for a few tenants. The developers did not configure a specific monitor for each of the tenants but monitored all tenants as a whole. As a result, the monitor was not sensitive enough and only triggered when most of the tenants' requests failed. Nevertheless, iPACK continuously runs and could still correlate the alert with the resultant tickets after the alert was finally fired. In this way, iPACK can potentially discover such under-monitoring cases and guide the configuration of monitors to improve system reliability [59]. Fortunately, such cases (tickets submissions before alerts) are rare in Azure with comprehensive monitoring according to our study (Section II-C).

VI. Related Work

A. Incident Analysis

Researchers have devoted sustained efforts on empirical studies [60], [61], [3], [4], [23] of cloud incidents in the last few years. Gunawi et al. [60] discussed why outages still take place in cloud environments by analysing headline news and public postmortem reports of 32 popular Internet services. Huang et al. [61] discussed their experiences with gray failure in production cloud-scale systems and demonstrated its broad scope and consequences. Chen et al. [23] presented a comprehensive study on how alerts and incidents are managed in large-scale public cloud. Cloud alerts are notoriously blamed for its great volume. In general, there are two threads of studies proposed towards resolving the challenge. The major thread aims to correlate alerts that are caused by the same incident [18][51][6]. Given a large number of alerts happening, Chen et al. [37] empirically found that only a small portion of alerts matters and proposed to prioritize alerts based on historical data. Chen et al. [32][62] proposed to predict the link between two alerts by combining alert textual information and the topology information among alerts (i.e., the topology of components that generate these alerts). These studies either require experts' manual annotations [24][32] or precise system topology [6][19]. Differently, we propose GIP, which does not require such labels or prior knowledge to identify alert-alert relations. We further leverage the alert-alert relations to aggregate tickets for efficient processing and management.

B. Issue Report Analysis

Issue reports, including app reviews, user feedback, bug reports, test reports, GitHub issues, support tickets, etc., are crucial for service providers to gain a better understanding of their customers' experiences. A large body of research has been devoted to the analysis of issue reports, covering topics such as duplicate bug reports detection [41][43][44][13], emerging issue detection [46] [10][47] bug reproduction [63][64], bug report summarization [65][66] and empirical studies [67][68][69].

Most existing studies focus on natural language text information such as titles and descriptions. In addition, some latest attempts [70][71][72] proposed to jointly consider multi-modality features, e.g., text and images (e.g., app screenshots), which has become a recent hot trend in the research direction. Different from these studies that purely focus on the customer-side issue report information, in this work, we also consider ongoing alerts and incidents in the complex cloud system. We aim to bridge the cloud alerts with cloud users' tickets to facilitate efficient ticket processing.

VII. Threats to Validity

External Validity. The study's object is the primary external threat. The data was collected from Azure, as there is no publicly available dataset containing customer tickets and a large number of alerts. However, Azure is a world-leading cloud provider with a vast scale. The data covers a broad range of services from various regions (Section IV-A1). Hence, the evaluation in Azure should be representative and convincing. Furthermore, iPACK leverages the common features provided by the most popular cloud vendors (Section II-A), making it capable of generalizing to similar cloud systems, potentially benefiting cloud customers globally.

Internal Validity. Implementation and parameter setting are the main internal threats to validity. For implementation, the baseline approaches are not open-sourced, so we re-implemented them by following the original papers closely. To reduce the implementation threat, we leveraged several mature libraries for implementing the core algorithms (Section IV-A2). Both the proposed and baseline methods underwent peer code review. For parameter setting, we tuned all methods through grid-search and chose the best results.

VIII. Conclusion

This paper tackles the problem of aggregating duplicate customer support tickets for cloud systems. Previous solutions that mainly rely on customer-side information (i.e., textual similarity between tickets) are sub-optimal for tickets of large-scale cloud systems. The main cause is the complexity of cloud systems that consist of many inter-dependent services, where the customers may experience distinct issues even though they are affected by the same incident. To overcome this limitation, we propose iPACK to leverage alerts of cloud systems to facilitate ticket aggregation. Specifically, we propose graph-based incident profiling (GIP) to model alert-alert relations and attentive interaction network (AIN) to model alert-ticket relations, respectively. In this way, we can aggregate the tickets that are linked to the same incident (linked alerts) even though they carry dissimilar semantics. We evaluate iPACK based on three datasets collected from the real-world production environment in a large-scale cloud vendor, Azure. iPACK achieves the F1 score of 0.871$\sim$0.935 and outperforms state-of-the-art methods by 12.4%$\sim$31.2% across the three datasets.

For future work, we will deploy iPACK to more services in Azure and conduct rigorous user studies among the support engineers to understand the usefulness in accelerating support tickets. In addition, we plan to extend iPACK with the ability to conduct root cause analysis based on the correlated alerts.

IX. Data Availability

The ticket data used in this work is collected from a real-world cloud vendor, which is highly confidential and contains a lot of personally identifiable information (PII). To protect customers' privacy, we decide not to release the original dataset. However, to facilitate the community to benefit from our work, we release the source code of iPACK together with some *synthetic data samples* on Github (https://github.com/OpsPAI/iPACK.git).

X. Acknowledgement

The work described in this paper was supported by the Research Grants Council of the Hong Kong Special Administrative Region, China (No. CUHK 14206921 of the General Research Fund) and Australian Research Council (ARC) Discovery Projects (DP200102940, DP220103044).

REFERENCES

[1] "Aws support plan comparison." [Online]. Available: https://aws.amazon.com/premiumsupport/plans/

[2] "Support scope and responsiveness." [Online]. Available: https://azure.microsoft.com/en-us/support/plans/response/

[3] 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.

[4] 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.

[5] 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.

[6] 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 42nd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2020, pp. 162–171.

[7] M. A. Shahid, N. Islam, M. M. Alam, M. M. Su'ud, and S. Musa, "Towards resilient method: An exhaustive survey of fault tolerance methods in the cloud computing environment," *Comput. Sci. Rev.*, vol. 40, p. 100398, 2021.

[8] C. Sun, D. Lo, X. Wang, J. Jiang, and S.-C. Khoo, "A discriminative model approach for accurate duplicate bug report retrieval," in *Proceedings of the 32nd International Conference on Software Engineering (ICSE)*, 2010, pp. 45–54.

[9] C. Sun, D. Lo, S.-C. Khoo, and J. Jiang, "Towards more accurate retrieval of duplicate bug reports," in *Proceedings of the 26th International Conference on Automated Software Engineering (ASE)*, 2011, pp. 253–262.

[10] W. Zheng, H. Lu, Y. Zhou, J. Liang, H. Zheng, and Y. Deng, "ifeedback: exploiting user feedback for real-time issue detection in large-scale online service systems," in *Proceedings of the 34th International Conference on Automated Software Engineering (ASE)*, 2019, pp. 352–363.

[11] X. Yang, D. Lo, X. Xia, L. Bao, and J. Sun, "Combining word embedding with information retrieval to recommend similar bug reports," in *Proceedings of the 27th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2016, pp. 127–137.

[12] A. Budhiraja, K. Dutta, R. Reddy, and M. Shrivastava, "Dwen: deep word embedding network for duplicate bug report detection in software repositories," in *Proceedings of the 40th International Conference on Software Engineering: companion proceeedings (ICSE-C)*, 2018, pp. 193–194.

[13] A. Budhiraja, R. Reddy, and M. Shrivastava, "Lwe: Lda refined word embeddings for duplicate bug report detection," in *Proceedings of the 40th International Conference on Software Engineering: companion proceeedings (ICSE-C)*, 2018, pp. 165–166.

[14] M. Haering, C. Stanik, and W. Maalej, "Automatically matching bug reports with related app reviews," in *Proceedings of the 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 970–981.

[15] A. Lazar, S. Ritchey, and B. Sharif, "Generating duplicate bug datasets," in *Proceedings of the 11th working conference on mining software repositories (MSR)*, 2014, pp. 392–395.

[16] A. Lamkanfi, J. Pérez, and S. Demeyer, "The eclipse and mozilla defect tracking dataset: a genuine dataset for mining bug information," in *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR)*, 2013, pp. 203–206.

[17] 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 *Proceedings of the 36th International Conference on Automated Software Engineering (ASE)*, 2021, pp. 653–665.

[18] 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.

[19] 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 *Proceedings of the 36th International Conference on Automated Software Engineering (ASE)*, 2021, pp. 430–442.

[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] "Introduction to alerting." [Online]. Available: https://cloud.google.com/monitoring/alerts

[23] 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.

[24] 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.

[25] T. Yang, J. Shen, Y. Su, X. Ren, Y. Yang, and M. R. Lyu, "Characterizing and mitigating anti-patterns of alerts in industrial cloud systems," in *Proceedings of the 52st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2022.

[26] "Aws support." [Online]. Available: https://aws.amazon.com/premiumsupport/

[27] "Google support hub." [Online]. Available: https://cloud.google.com/support-hub

[28] Z. Li, H. Li, T.-H. Chen, and W. Shang, "Deeplv: Suggesting log levels using ordinal based neural networks," in *Proceedings of the 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 1461–1472.

[29] Wikipedia, "Curse of dimensionality." [Online]. Available: http://en.wikipedia.org/w/index.php?title=Curseofdimensionality

[30] J. Zhu, S. He, J. Liu, P. He, Q. Xie, Z. Zheng, and M. R. Lyu, "Tools and benchmarks for automated log parsing," in *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2019, pp. 121–130.

[31] P. He, J. Zhu, Z. Zheng, and M. R. Lyu, "Drain: An online log parsing approach with fixed depth tree," in *Proceedings of the IEEE International Conference on Web Services (ICWS)*. IEEE, 2017, pp. 33–40.

[32] 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.

[33] Wikipedia, "Pointwise mutual information." [Online]. Available: http://en.wikipedia.org/w/index.php?title=Pointwisemutualinformation

[34] H. Qin, Y. Tian, and Y. Song, "Relation extraction with word graphs from n-grams," in *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2021, pp. 2860–2868.

[35] L. Yao, C. Mao, and Y. Luo, "Graph convolutional networks for text classification," in *Proceedings of the AAAI conference on artificial intelligence (AAAI)*, 2019, pp. 7370–7377.

[36] V. Satopaa, J. Albrecht, D. Irwin, and B. Raghavan, "Finding a" kneedle" in a haystack: Detecting knee points in system behavior," in *2011 31st international conference on distributed computing systems workshops*. IEEE, 2011, pp. 166–171.

[37] 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.

[38] Wikipedia, "Component (graph theory)." [Online]. Available: http://en.wikipedia.org/w/index.php?title=Component(graphtheory)

[39] S. Rendle, "Factorization machines," in *2010 IEEE International conference on data mining*. IEEE, 2010, pp. 995–1000.

[40] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of (NAACL-HLT)*. Association for Computational Linguistics, 2019, pp. 4171–4186.

[41] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun, "An approach to detecting duplicate bug reports using natural language and execution information," in *Proceedings of the 30th international Conference on Software Engineering (ICSE)*, 2008, pp. 461–470.
[42] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *Proceedings of the 3rd International Conference on Learning Representations (ICLR)*, 2015.
[43] A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, D. Lo, and C. Sun, "Duplicate bug report detection with a combination of information retrieval and topic modeling," in *Proceedings of the 27th International Conference on Automated Software Engineering (ASE)*, 2012, pp. 70–79.
[44] J. Zhou and H. Zhang, "Learning to rank duplicate bug reports," in *Proceedings of the 21st ACM International Conference on Information and Knowledge Management (CIKM)*, 2012, pp. 852–861.
[45] O. Chaparro, J. M. Florez, U. Singh, and A. Marcus, "Reformulating queries for duplicate bug report detection," in *Proceedings of the 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2019, pp. 218–229.
[46] C. Gao, J. Zeng, M. R. Lyu, and I. King, "Online app review analysis for identifying emerging issues," in *Proceedings of the 40th International Conference on Software Engineering (ICSE)*, 2018, pp. 48–58.
[47] C. Gao, W. Zheng, Y. Deng, D. Lo, J. Zeng, M. R. Lyu, and I. King, "Emerging app issue identification from user feedback: Experience on wechat," in *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2019, pp. 279–288.
[48] H. Liu, M. Shen, J. Jin, and Y. Jiang, "Automated classification of actions in bug reports of mobile apps," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2020, pp. 128–140.
[49] H. Wu, W. Deng, X. Niu, and C. Nie, "Identifying key features from app user reviews," in *Proceedings of the 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 922–932.
[50] Wikipedia, "Hierarchical Clustering." [Online]. Available: https://en.wikipedia.org/wiki/Hierarchical_clustering
[51] 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.
[52] "Allennlp." [Online]. Available: https://allenai.org/allennlp
[53] "Scikit-learn." [Online]. Available: https://scikit-learn.org/
[54] "Gensim." [Online]. Available: https://radimrehurek.com/gensim/
[55] "Huggingface." [Online]. Available: https://github.com/huggingface/transformers
[56] W. M. Rand, "Objective criteria for the evaluation of clustering methods," *Journal of the American Statistical association*, vol. 66, no. 336, pp. 846–850, 1971.
[57] E. Achtert, S. Goldhofer, H.-P. Kriegel, E. Schubert, and A. Zimek, "Evaluation of clusterings–metrics and visual support," in *Proceedings of the 28th International Conference on Data Engineering (ICDE)*, 2012, pp. 1285–1288.
[58] A. J. Gates and Y.-Y. Ahn, "The impact of random models on clustering similarity," *Journal of Machine Learning Research (JMLR)*, vol. 18, pp. 1–28, 2017.
[59] Y. Li, X. Zhang, S. He, Z. Chen, Y. Kang, J. Liu, L. Li, Y. Dang, F. Gao, Z. Xu *et al.*, "An intelligent framework for timely, accurate, and comprehensive cloud incident detection," *ACM SIGOPS Operating Systems Review*, vol. 56, no. 1, pp. 1–7, 2022.
[60] 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.
[61] 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.
[62] 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.
[63] Y. Zhao, T. Yu, T. Su, Y. Liu, W. Zheng, J. Zhang, and W. G. Halfond, "Recdroid: automatically reproducing android application crashes from bug reports," in *Proceedings of the 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 128–139.
[64] Y. Cao, H. Zhang, and S. Ding, "Symcrash: Selective recording for reproducing crashes," in *Proceedings of the 29th International Conference on Automated software engineering (ASE)*, 2014, pp. 791–802.
[65] S. Rastkar, G. C. Murphy, and G. Murray, "Automatic summarization of bug reports," *IEEE Transactions on Software Engineering (TSE)*, vol. 40, no. 4, pp. 366–380, 2014.
[66] X. Li, H. Jiang, D. Liu, Z. Ren, and G. Li, "Unsupervised deep bug report summarization," in *Proceedings of the 26th International Conference on Program Comprehension (ICPC)*. IEEE, 2018, pp. 144–14 411.
[67] B. Kucuk and E. Tuzun, "Characterizing duplicate bugs: An empirical analysis," in *Proceedings of the International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2021, pp. 661–668.
[68] W. Zou, D. Lo, Z. Chen, X. Xia, Y. Feng, and B. Xu, "How practitioners perceive automated bug report management techniques," *IEEE Transactions on Software Engineering (TSE)*, vol. 46, no. 8, pp. 836–862, 2018.
[69] W. Ma, L. Chen, X. Zhang, Y. Zhou, and B. Xu, "How do developers fix cross-project correlated bugs? a case study on the github scientific python ecosystem," in *Proceedings of the 39th International Conference on Software Engineering (ICSE)*, 2017, pp. 381–392.
[70] J. He, L. Xu, M. Yan, X. Xia, and Y. Lei, "Duplicate bug report detection using dual-channel convolutional neural networks," in *Proceedings of the 28th International Conference on Program Comprehension (ICPC)*, 2020, pp. 117–127.
[71] D. Liu, Y. Feng, X. Zhang, J. Jones, and Z. Chen, "Clustering crowdsourced test reports of mobile applications using image understanding," *IEEE Transactions on Software Engineering (TSE)*, 2020.
[72] N. Cooper, C. Bernal-Cárdenas, O. Chaparro, K. Moran, and D. Poshyvanyk, "It takes two to tango: Combining visual and textual information for detecting duplicate video-based bug reports," in *Proceedings of the 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 957–969.