CONAN: Diagnosing Batch Failures for Cloud Systems

Liqun Li[‡], Xu Zhang[‡], Shilin He[‡], Yu Kang[‡], Hongyu Zhang[o], Minghua Ma[‡], Yingnong Dang[†], Zhangwei Xu[†], Saravan Rajmohan[◇], Qingwei Lin[‡*], Dongmei Zhang[‡]
[‡]*Microsoft Research*, [†]*Microsoft Azure*, [◇]*Microsoft 365*, [o]*The University of Newcastle*

***Abstract*—Failure diagnosis is critical to the maintenance of large-scale cloud systems, which has attracted tremendous attention from academia and industry over the last decade. In this paper, we focus on diagnosing *batch failure*s, which occur to a batch of instances of the same subject (e.g., API requests, VMs, nodes, etc.), resulting in degraded service availability and performance. Manual investigation over a large volume of high-dimensional telemetry data (e.g., logs, traces, and metrics) is labor-intensive and time-consuming, like finding a needle in a haystack. Meanwhile, existing proposed approaches are usually tailored for specific scenarios, which hinders their applications in diverse scenarios. According to our experience with Azure and Microsoft 365 – two world-leading cloud systems, when batch failures happen, the procedure of finding the root cause can be abstracted as looking for *contrast* patterns by comparing two groups of instances, such as failed vs. succeeded, slow vs. normal, or during vs. before an anomaly. We thus propose CONAN, an efficient and flexible framework that can automatically extract contrast patterns from contextual data. CONAN has been successfully integrated into multiple diagnostic tools for various products, which proves its usefulness in diagnosing real-world batch failures.**

I. Introduction

Failures of cloud systems (e.g., Azure[8], AWS[7], and GCP [6]) could notoriously disrupt online services and impair system availability, leading to revenue loss and user dissatisfaction. Hence, it is imperative to rapidly react to and promptly diagnose the failures to identify the root cause after their occurrence [10], [54]. In this work, we focus on diagnosing *batch failure*, which is a common type of failure widely found in cloud systems. A batch failure is composed of many individual instances of a certain subject (e.g., API requests, computing nodes, VMs), typically within a short time frame. For example, a software bug could cause thousands of failed requests [42]. In cloud systems, batch failures tend to be severe and usually manifest as *incidents* [20], which cause disruption or performance degradation.

Batch failure in cloud systems could be caused by reasons including software and configuration changes [32], [48], power outages [1], disk and network failures [43], [31], etc. To diagnose batch failures, engineers often retrieve and carefully examine the *contextual data* of the instances, such as their properties (e.g., version or type), run-time information (e.g., logs or traces), environment and dependencies (e.g., nodes or routers), etc. Contextual information can often be expressed in the form of *attribute-value pairs* (AVPs) denoted as Attribute-Name="Value". Let's say a request emitted by App APP1 is served by a service of APIVersion V1 hosted on Node N1. Then, the contextual information of this specific request can be expressed with 3 AVPs, i.e., App="APP1", APIVersion="V1", and Node="N1". If a batch of failed API requests occurs due to the incompatibility between a specific service version (say APIVersion="V1") and a certain client application (say App="APP1"). Then, the combination of {APIVersion="V1", App="APP1"} is what engineers aim to identify during failure diagnosis.

Identifying useful AVPs from the contextual data is often "case-by-case" based on "expert knowledge" for "diverse" scenarios according to our interview with engineers. Fortunately, a natural principle usually followed by engineers is to compare two groups of instances, such as failed vs. succeeded, slow vs. normal, or during vs. before an anomaly, etc. The objective is to look for a set of AVPs that can significantly differentiate the two groups of instances. We call such a set of AVPs a *contrast pattern*. However, manual examination of contrast patterns, which requires comparison over a large volume of high-dimensional contextual data, is labor-intensive, and thus cannot scale up well. In one scenario (Sec. V-C), it takes up to 20 seconds to visualize only one data pivot, while there are hundreds of thousands of AVP combinations. Recently, many failure diagnosis methods have been proposed (summarized in Table I). These approaches have been shown to be useful in solving a variety of problems. However, they are rigid in adapting to new scenarios, so developers have to re-implement or even re-design these approaches for new scenarios.

In this paper, we summarize common characteristics from diverse diagnosis scenarios, based on which we propose a unified data model to represent various types of contextual data. We propose a framework, namely CONAN, to automatically search for contrast patterns from the contextual data. CONAN first transforms diverse input data into a unified format, and then adopts a meta-heuristic search method [17] to extract contrast patterns efficiently. Finally, a consolidation process is proposed, by considering the concept hierarchy in the data, to produce a concise diagnosis result.

The main advantage of CONAN over existing methods [51], [10], [44], [40] is that it can be applied flexibly to various scenarios. CONAN supports diagnosing both availability and performance issues, whereas existing work as summarized in Table I only supports one of them. In CONAN, it is flexible

*Qingwei Lin is the corresponding author of this work.

to measure the significance of patterns in differentiating the two instance groups. For example, CONAN can find patterns that are prevalent in abnormal instances but rare in normal ones, or patterns with a significantly higher latency only during the incident time. Moreover, CONAN supports multiple data types including tabular telemetry data and console logs. We have integrated CONAN in diagnostic tools used by Azure and Microsoft 365 – two world-leading cloud systems. In the last 12 months, CONAN had helped diagnose more than 50 incidents from 9 scenarios. Its advantages have been affirmed in real-world industrial practice, greatly saving engineers' time and stopping incidents from impacting more services and customers.

To summarize, our main contributions are as follows:

- To the best of our knowledge, we are the first to unify the diagnosis problem in different batch failure scenarios into a generic problem of extracting contrast patterns.
- We propose an efficient and flexible diagnosis framework CONAN, which can be applied to diverse scenarios.
- We integrate CONAN in a variety of typical real-world products and share our practices and insights in diagnosing cloud batch failures from 5 typical scenarios.

The rest of this paper is organized as follows. In Sec. II, we introduce the background of batch failure and demonstrate an industrial example. Sec. III presents the data model and problem formulation. The proposed framework CONAN and its implementation are described in Sec. IV. We show real-world applications in Sec. V. At last, we discuss CONAN in Sec. VI, summarize related work in Sec. VIII, and conclude the paper in Sec. IX.

II. Background and Motivating Example

For a cloud system, when a batch of instances (e.g., requests, VMs, nodes) fail, the monitoring infrastructure can detect them immediately, create an incident ticket, and send alerts to on-call engineers to initiate the investigation. Although various automated tools (e.g., auto-collecting exceptional logs and traces) have been built to facilitate the diagnosis process, *how to quickly diagnose the failure* remains the bottleneck, especially due to the excessive amount of telemetry data.

A. A Real-world Scenario: Safe Deployment

The Exchange service is a large-scale online service in Microsoft 365, which is responsible for message hosting and management. To reduce the impact caused by defective software changes, the service employs a safe deployment mechanism, similar to what is described in [5], [16], [32]. That is, a new build version needs to go through several release phases before meeting customers. Due to the progressive delivery process, multiple build versions would co-exist in the deployment environment, as shown in Fig. 1, where nodes with different colors are deployed with different versions. Client applications interact via REST API requests.

A **batch of failed requests** would trigger an incident. The key question for safe deployment is: *Is the problem caused by a recent deployment or by ambient noise such as a hardware failure or a network issue*? To answer this question, a snapshot of **contextual data** during the incident is collected. The data typically contains tens of millions of instances of requests for this large-scale online service. Each instance represents a failed or succeeded request. A request has many associated attributes. To give a few examples, it has the APP attribute that denotes the client application which invokes this API, and the APIVersion that shows the server-side software build version of the invoked API. There are attributes, such as Cluster, Availability Group (AG), and Node, that describe the location where requests are routed and served[1]. We aim to answer the aforementioned question based on the collected request data.

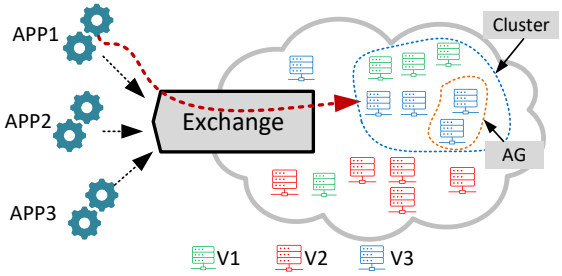

Fig. 1. Multiple build versions co-exist during service deployment life-cycle. Nodes with different colors correspond to different build versions.

The idea behind their existing practice is intuitive. If the incident is caused by a build version, then its failure rate should be higher than other versions. For example, if APIVersion V3 has a significantly higher failure rate than V1 and V2, the production team suspects that this is a deployment issue caused by V3. Then, engineers deep dive into the huge code base of the suspected version. It could waste a lot of time before they can confirm that this is not a code regression if the initial direction given by the safe deployment approach is wrong.

This approach sheds light on diagnosing batch failures by comparing the occurrence of an attribute (APIVersion in our example) among two categories of instances, i.e., failed and succeeded requests. However, it has two drawbacks. First, it only concerns the APIVersion attribute while neglecting other attributes. A large number of failed requests emerging on a specific version might be caused by problems such as node or network failures, resulting in misattributing a non-deployment issue to a build version problem. Second, a batch failure could be caused by a combination of multi-attributes containing a certain version, as will be discussed in Sec. V-A. Then, the failure rate of any single version might not be significantly higher than other versions, leading to misidentification of the build version problem. Either way would lead to a prolonged diagnosis process.

The production team could certainly enrich their method by adding heuristic rules to cover more cases. However, it is ad-hoc and error-prone. In summary, we need a systematic solution to diagnose batch failures for safe deployment.

[1]One cluster consists of multiple AGs, and one AG is composed of 16 nodes.

TABLE I
COMMONALITIES IN DIFFERENT DIAGNOSIS WORK

Failed instances	Contextual data	Contrast class	diagnostic output
Requests [42], [12], [52]	Attributes and components on the critical paths or call graphs	Slow vs. normal requests or failure vs. successful requests	Attribute combinations, e.g., {Cluster= "PrdC01", API= "GET"}, problematic components, or structural mutations
Software crash reports [41], [19], [38], [33], [47]	Attributes or traces, e.g., OS, modules, navigation logs, events, etc.	Crash vs. non-crash or different types of crashes	Attribute combinations, e.g., {OS="Android", Event="upload"} or event sequence or functions
OS events [49]	Driver function calls and status along the traces	Slow vs. normal event traces	Combination of drivers and functions, e.g., {fv.sys!Func1, fs.sys!Func2, se.sys!Func3}
Virtual disks [51]	VM, storage account, and network topology	Failure vs. normal disks	Problematic component, e.g., a router or a storage cluster
Customer reports [35]	Attributes, e.g., country, feature, etc.	Reports during anomaly vs. before anomaly	Attribute combinations, e.g., {Country="India", Feature="Edit"}
System operations [39], [14]	System logs or attributes, e.g., server, API, etc.	Slow vs. normal operations	Attribute combinations, e.g., {Latency ¿ 568ms, Region="America"} or a set of indicative logs
System KPIs [53], [25]	System logs	Logs during vs. before the KPI anomaly	A set of indicative logs

III. BATCH FAILURE DIAGNOSIS

In this section, we first summarize commonalities among the practices of diagnosing batch failures. We then abstract the data model and formulate the failure diagnosis problem as a problem of identifying contrast patterns.

A. Commonalities in Scenarios

The first step in solving the batch failure diagnosis problem is to summarize the commonalities among diverse scenarios. However, the circumstances of the failures can appear so different that it is challenging to determine their commonalities. We have carefully analyzed the various cases encountered in practice, such as the safe deployment case introduced previously, and thoroughly reviewed existing research work. Table I presents a summary of literature studies on failure diagnosis. These studies focus on notably different scenarios, but they share the following commonalities:

- They are all failures that affect **a batch of instances** of the same subject (such as requests, software crashes, disks, and so on), instead of on a single or a handful of instances.
- Each failure instance has a collection of **attributes** that describe the context in which it occurred.
- Besides the instances involved in each batch failure (called **Target class**), we can always find another set of instances with different statuses or performance levels (called **Background class**) for comparison.
- The diagnosis output, based on the instances and their attributes, could be represented as **a combination of attribute-value pairs** (AVPs).

In our motivating scenario, the background class of instances are the requests that were successfully processed during the incident time. When the batch failure is about performance issues, such as high latency in request response time, we take the requests before the latency surge as the background class. By comparing the attributes of two sets of instances, we can identify patterns, namely contrast patterns, which can help narrow down the search for the root cause of the failure.

The commonalities in data characteristics and diagnosis process suggest that a generic approach could be effective in addressing the batch failure diagnosis problem. Since each scenario may look quite different, the framework must be flexible enough to benefit both current and future scenarios.

B. Contrast Pattern Identification

Contrast patterns in two contrast classes often demonstrate statistically significant differences. For instance, one pattern is more prevalent in the target class than the background class, or instances of a pattern have a higher average latency in the target class than in the background class.

The contrast pattern extraction can thus be formulated as a search problem. To achieve so, an *objective function* should be defined first, and then maximized during the following search process. We denote the objective function as $f_{B,T}(p)$, which quantitatively measures the difference of the pattern p in the two classes $\{\text{B, T}\}$. B and T stand for background class and target class, respectively. The goal of the search process is to find the pattern $\hat{p}$ which maximizes the objective function, i.e.,

$$\hat{p} = \arg\max_{p} f_{B,T}(p) \tag{1}$$

The objective function can vary across different diagnosis scenarios, as will be demonstrated in the practical examples in Sec. V. We now show the objective function, in Eq. (2), for the safe deployment example in Sec. II-A. The objective function is defined as the proportion difference between the failed and succeeded requests of any specific pattern p.

$$f_{B,T}(p) = \frac{|S_T(p)|}{|S_T|} - \frac{|S_B(p)|}{|S_B|} \tag{2}$$

$|\cdot|$ denotes the number of instances, $S_T(p)$ and $S_B(p)$ are target-class and background-class instances, respectively, which contain the pattern p. The intuition is that the pattern p should be more prevalent in the failed requests than in the succeeded requests.

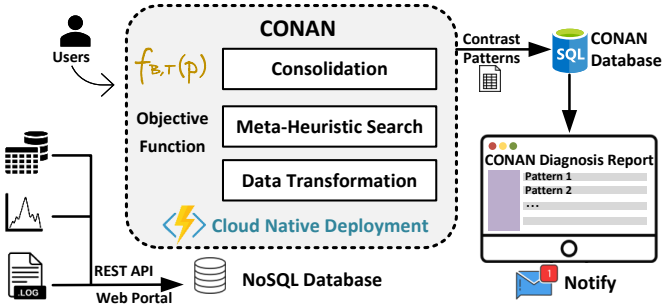

Fig. 2. An Overview of CONAN.

IV. The Proposed System

Several requirements are imposed for designing a generic framework for batch failure diagnosis. It needs to support various input data, search contrast patterns efficiently, and provide easy-to-interpret results. To fulfill these requirements, we propose a batch failure diagnosis framework, namely CONAN. Fig. 2 shows an overview of our system which consists of three components.

- Data transformation: We convert input data into our unified data model, i.e., instances. Each instance is represented with a set of AVPs, a class label, and optionally its metric value (e.g., latency).
- Meta-heuristic search: Following a user-specified objective function, contrast patterns are extracted by employing a meta-heuristic search algorithm.
- Consolidation: The search could incur duplication, i.e., multiple patterns describing the same group of instances. We design rules to consolidate the final output patterns to make them concise and easy to interpret.

In most scenarios, the output patterns are consumed by engineers or system operators. They may perform follow-up tasks to identify the actual root cause for triage, mitigation, and problem-fixing.

A. Data Transformation

In this section, we introduce the practices of transforming diverse diagnosis data into a unified format. An instance is an atomic object on which our diagnosis framework is performed. For example, in the safe deployment scenario, the diagnosis aims to find out why a batch of requests fail, and a request is thus treated as one instance. Similarly, as depicted in Table I, an instance could be a software crash report, an OS event, a virtual disk disconnection alert, a customer report, etc.

Attribute-value pair (AVP) is the data structure to denote the contextual data for diagnosis. In practice, there could be a large number of attributes. These attributes are usually scoped based on engineer experience to avoid involving unnecessary attributes or missing critical ones. For instance, we only care about the Client Application, APIVersion, Node, etc., whose issues can directly result in a request success rate drop for safe deployment. As batch failures could occur from time to time, the attributes may be adjusted gradually. In the beginning, engineers obtain a list of attributes based on their knowledge. In subsequent practice, new attributes are added or existing ones are removed depending on the situation. One can refer to the Contextual data column in Table I for typical attributes chosen for various batch failure diagnosis tasks.

A contrast class is assigned for each instance, i.e., target class and background class. The target class labels instances of our interest. For scenarios such as safe deployment, each instance has its status, namely, success or failure, then the status can naturally serve as the contrast class label. Sometimes, only "failed" instances are collected and the diagnosis purpose is to find out why a sudden increase of failures occurs (e.g., an anomaly). In this scenario, temporal information is used to decide the contrast class. We assign the target class to instances occurring during the anomaly duration and the background class to instances before the anomaly period. We shall present one such example in Sec. V-D.

Multiple attributes may form a hierarchical relationship. In our safe deployment example in Sec. II-A, we say Node is a low-level attribute compared to AG (Availability Group) because one AG contains multiple Nodes. Similarly, Node is also a lower-level attribute to APIVersion, as one APIVersion is typically deployed to multiple nodes while one node has only one APIVersion. For convenience, we denote the hierarchical relationship in a chain of attributes, e.g., Node $\longrightarrow$ AG $\longrightarrow$ Cluster. The chain starts from a low-level attribute (left side) and ends at a high-level attribute (right side). The hierarchical chains can be automatically mined from the input data because high-level attributes and their low-level attributes form one-to-many relationships, e.g., one AG corresponds to multiple nodes. CONAN provides a tool to analyze the hierarchical relationship in the input data for users. The hierarchy chains are then used for subsequent steps.

B. Meta-heuristic Search

To search for the pattern as desired, a straightforward way is to evaluate every possible pattern exhaustively with the objective function. However, the method is clearly computationally inefficient. Though we can explore all combinations of 1 or 2 AVPs in a brute-force way, we cannot rigidly limit the pattern length in practice. Thus, contrast pattern extraction desires a more systematic search algorithm.

In this work, we adopt a meta-heuristic search [17] framework but customized it especially to mine the contrast pattern. Compared to existing pattern mining [41], [19], [33] or model interpretation [51], [14] methods, heuristic search is more flexible to tailor for different scenarios. Meta-heuristic search is an effective combinatorial optimization approach. The input to the search framework is the data prepared as discussed in the above section. The output is a list of contrast patterns optimized toward high objective function values. The search framework features the following components:

- A contrast pattern list, denoted as L_c.
- A current in-search pattern, denoted as p.
- A set of two search operations (i.e., ADD and DEL).
- An objective function $f_{B,T}(p)$ that evaluates a pattern p.

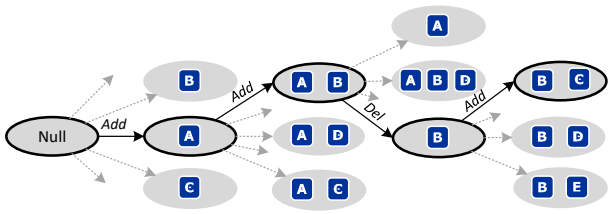

Fig. 3. Illustration of the search process where each uppercase character represents an AVP. In each step, the current pattern p (with black line color) randomly selects an operation (ADD or DEL) to transit to a new pattern. Patterns along the search path with the highest objective function values are kept in the contrast pattern list L_c.

The search process starts from an empty pattern $Null$ and goes through many iterations to find patterns optimizing towards the objective function, as illustrated in Fig. 3.

1) The Search Process: In each iteration, we randomly apply an operation to the current pattern to generate a new pattern. ADD means that we add an AVP to the current pattern; DEL means that we delete one AVP from the current pattern. When adding an AVP, we avoid the AVPs with attributes already presented in the current pattern. Once we have a new pattern, we use the objective function to evaluate its score. We maintain the best contrast patterns found in a fixed-size list L_c, which is the algorithm output. If the list L_c is not full or the score is higher than the minimum score of stored patterns in L_c, we add or update the pattern to L_c.

The search process is essentially finding desired patterns w.r.t the objective function, as defined in Eq. (2) for our example scenario. The search ends when a static or dynamic exit criterion is satisfied. In static criteria, we could end the search process after a predefined number of steps or a time interval. In dynamic criteria, the search stops if the pattern list L_c does not update for certain steps.

One advantage of meta-heuristic search is that it endorses the early-stopping mechanism naturally. We can end the process early as required and meanwhile obtain reasonably great results, which is very helpful in diagnosis scenarios under hard time constraints. Besides, we could explicitly limit the length of the pattern during the search process. Once the current pattern reaches the max length, we prevent the search process from ADD operations.

2) The Scoring Function: When applying the ADD operation to the current pattern, simply choosing a random AVP is inefficient. Instead, an AVP should be added if it could benefit the new pattern, e.g., achieving a higher objective function score. In meta-heuristic search [55], the algorithm typically explores the whole neighborhood of the current state to find the next state that maximizes the objective function, which however is not computationally affordable due to the huge pattern space. As an approximation, we introduce a *scoring function* [26], [46] $f_s(AVP)$ for each AVP. Specifically, we inherit the objective function and calculate the score by

$$f_s(AVP) = f_{B,T}(\{AVP\}) \tag{3}$$

where $\{AVP\}$ is a pattern with only one attribute-value pair. When we need to pick an AVP to add to the current pattern, we choose the one with the highest score. Though the pattern space is large due to combination explosion, the number of AVPs is much smaller. Therefore, we can calculate a lookup table for the score of each AVP beforehand. In our implementation, we adopt the BMS (Best from Multiple Selections) mechanism [18], which selects the top few AVPs with the highest scores and then samples one from them with probability proportional to their scores.

It is easy for a search algorithm to be stuck in a local search space if we add AVPs in a purely greedy fashion, i.e., always pick the AVP with the highest score. We thus maintain a tabu list, inspired by the Tabu search algorithm [21], which tracks recently explored AVPs to avoid revisiting them in a cycling fashion. The tabu list size plays a key role in balancing the exploration and exploitation of the search process.

Recall that we have mined the hierarchy chains in Sec. IV-A. When the current pattern contains an AVP of a low-level attribute (e.g., Node), it makes no sense to add an AVP of high-level attributes (e.g., AG or Cluster). We thus enforce this rule during the search process to reduce the search space.

C. Consolidation

The output of the meta-heuristic search (Sec. IV-B) is a list of patterns sorted by their objective function values in descending order. We identify two situations that could lead to pattern redundancy, which could cause confusion to the user. They are shown in the following examples:

- p_1: {App="APP1", APIVersion="V1"},
  p_2: {APIVersion="V1"}
- p_1: {Node="N01"}, p_2: {AG="AG01"}[2]

In each example, we have two patterns p_1 and p_2. We use $S(p_1)$ and $S(p_2)$ to denote the sets of instances containing p_1 and p_2, respectively. In the first example, p_2 is actually a subset of p_1 given each pattern is a set of AVPs. Thus, p_1 is describing a smaller set of instances compared to p_2, i.e., $S(p_1) \subseteq S(p_2)$. This is also the case for the second example due to the existence of the hierarchy chain: Node $\longrightarrow$ AG $\longrightarrow$ Cluster. Patterns composed of low-level attributes are more specific than patterns with high-level attributes.

Under both circumstances, it could cause confusion if both p_1 and p_2 are presented to the user. Taking the second situation as an example, users may wonder whether this is an AG-scale issue or only a single-node issue. Once we identify the redundancy situations, we apply rules to deduplicate and retain only the major contributor pattern in the result list. Specifically, we pick the pattern associated with the minimum number of instances if two patterns achieve comparable objective function scores, which provides more fine-grained hints to localizing the root cause.

D. System Implementation

We implement CONAN as a Python library for easy reuse and customization. We design the objective scoring function

[2]Assume Node "N01" resides in AG "AG01"

as a callback function, which can be specified according to the target scenario. CONAN also integrates multiple commonly-used objective functions for user selection. We abstract the search interface and provide our meta-search algorithm described in IV-B as a concrete implementation. Users thus can implement the interface with other search algorithms. At last, we provide a set of data processing tools to accommodate various data formats. The library is implemented based on Python 3.7 and will be open-sourced after paper publication with sample datasets.

We also pack CONAN as a cloud-native application in Azure as shown in Fig. 2. We adopt the serverless function service, which enables easy scalability and maintenance. In this application, we support both REST APIs and a GUI web portal. The former could be seamlessly integrated into production pipelines. Contextual data is prepared and stored in a NoSQL database. The diagnosis is carried out in an async fashion and the results could later be retrieved from a SQL database. On the other hand, users could manually upload some test datasets for ad-hoc explorations via the GUI web portal. It then shows a report when the diagnosis is done.

The raw input data can sometimes be huge. For instance, in the safe deployment case, the input data is a table of requests, where each row is a single request with various attribute values. It is common to see tens of millions of requests per hour, resulting in input files over 10 GB. It is computationally expensive to process large files. Therefore, we support an aggregate data format where rows with the same set of AVPs are grouped into one row, with an additional *Count* column. According to our evaluation, the aggregate format reduces the data size by up to 100x in our scenarios. For diagnosing performance issues, the raw data cannot be directly aggregated in the aforementioned way as each row has its metric value (e.g., the response latency). CONAN supports aggregating performance data into buckets [4]. For instance, requests with the same set of AVPs are distributed into buckets of $< 100ms$, $100 - 500ms$, $500 - 1000ms$, etc., and their counts.

V. Real-world Practices

In this section, we present several representative scenarios that are common across cloud systems. We describe the typical practices of applying CONAN, which consists of 4 steps: (1) define the instances and collect their attributes; (2) set up the two contrast classes and the objective function; (3) extract the attribute hierarchy chains; (4) apply CONAN to search for contrast patterns. We will introduce the scenarios one by one in the following sections.

A. Safe Deployment

The background of safe deployment has been discussed in Sec. II-A. The problem formulation is detailed in Sec. III-B. In this section, we describe how to apply CONAN and present its evaluation results.

a) Applying CONAN: Each request is regarded as an instance. The target class and background class are the failed requests and succeeded requests during the incident period, respectively. The objective function is defined in Eq. (2) which is the proportion difference of the pattern between the two classes.

Intuitively, if one pattern is common in the target class but rare in the background class, it is likely related to the root cause. The attribute hierarchy chains for the safe deployment scenario are extracted from data, as follows:

- Node $\longrightarrow$ AG $\longrightarrow$ Cluster
- Node $\longrightarrow$ OS
- Node $\longrightarrow$ APIVersion

The hierarchy chains are used for search and consolidation as introduced in Sec. IV-B and IV-C, respectively.

b) Results: We run CONAN side-by-side with the existing diagnosis tool (described in the existing practice in Sec. II-A) for about 3 months. For each detected incident, CONAN determines it as a deployment issue as long as the resulting pattern with the highest objective function score contains an AVP of APIVersion. Then we triage the incident to the corresponding team with the found APIVersion to verify its correctness. The results are shown in Table II.

TABLE II
SAFE DEPLOYMENT RESULTS

	TP	TN	FP	FN	Precision	Recall
Existing approach	12	0	15	2	44.4%	85.7%
CONAN	14	12	3	0	82.4%	100%

There were a total of 29 cases during the 3-month period. We can see all deployment issues and their corresponding problematic APIVersions were successfully captured by CONAN. In contrast, their existing approach missed two (i.e., 2 **F**alse **N**egatives) because it could not recognize multi-attribute problems combined with bad build versions. More than that, the main problem of the existing approach lies in its high **F**alse-**P**ositive rate. A total of 15 non-deployment issues were misidentified, most of which were due to AG or node-level hardware issues. We note that CONAN also had 3 False Positives, which were caused by issues such as configuration or network issues that happen to co-exist with the deployment process of certain build versions. CONAN can be further improved by involving more relevant attributes. Thanks to the high efficiency of our meta-heuristic search algorithm, CONAN takes $\sim$ 20s even for 10^8 data size.

Example: The Exchange service once encountered an increasing number of failed requests on a REST API at the early stage of integration with CONAN. Most failures were concentrated on a recently deployed API version (V1). The on-call engineers spent several hours investigating V1 but still could not find anything wrong. CONAN was manually triggered and a contrast pattern {App="APP1", APIVersion="V1"} was reported. With this information, engineers spent less than half an hour engaging the right response team. The code change in service version V1 caused a class casting failure in the event handler in the client APP1. That explained why most failed requests were emitted by APP1 to the specific APIVersion V1.

B. VM Live Migration Blackout Performance

a) Background: Live migration of VMs [3] is a common practice in Azure for various objectives such as ensuring higher availability by moving VMs from failing nodes to healthy ones, optimizing capacity by defragmentation, improving efficiency by balancing node utilization, and regular hardware/software maintenance. During the live migration process, a VM moves from its source node to a destination node when the VM is running except for a short period, namely *blackout*. The customer's tasks are frozen during the blackout period. So, it is critical to minimize this duration.

The blackout period of a live migration session consists of tens of components. The blackout duration is thus the sum of the execution time of all components. The Compute team logs end-to-end blackout durations as well as the time breakdown for each component. An incident is fired once there is significant growth momentum of the blackout duration.

b) Existing Practice: When an incident is reported, the Compute team collects the top 10 cases with the longest blackout durations. The engineer reviews the cases and takes various diagnosis actions. The blackout duration could be affected by many factors such as the VM type, node hardware, OS, etc. Therefore, the diagnosis is very challenging and time-consuming. It could take days to find out the actual root cause.

c) Applying CONAN: Each component has its execution time for each live migration session. To diagnose a performance issue, we define the target class and background class as components belonging to live migration sessions during and before the anomaly period, respectively. Each live migration session has attributes such as VM type and the node/cluster/OS properties of the source/destination node. Intuitively, we intend to discover the pattern where there is a clear rising trend of the execution time during the anomaly period. We therefore define the objective function as shown in Eq. (4). $P_{90\%}(\cdot)$ is a function to calculate the 90th percentile[3] of the execution time of a set of instances.

$$f_{B,T}(p) = P_{90\%}\left(S_T(p)\right) - P_{90\%}\left(S_B(p)\right) \tag{4}$$

d) Examples: We describe applying CONAN for two performance incidents:

Incident-1 One day a significant execution time increase was monitored. CONAN found that the pattern with the highest objective function value contained a subscription name, i.e., {Component="A", Cluster="c1", Subscription="Sub2"}. After some investigations, the engineer found that a thread from the VMs of this customer, who subscribed hundreds of VMs, blocked component A of the live migration process from obtaining the network status at the nodes, and thus, caused a significant delay. Figure 4 (left) shows the curves of the daily 90th percentile of the execution time of component A for all live migration processes (the solid line) as well as for 3 specific subscriptions (the dashed lines), respectively. We can clearly see the increase in the overall execution time is driven by that of "Sub2".

[3]We use the 90th percentile because it is also used for monitoring.

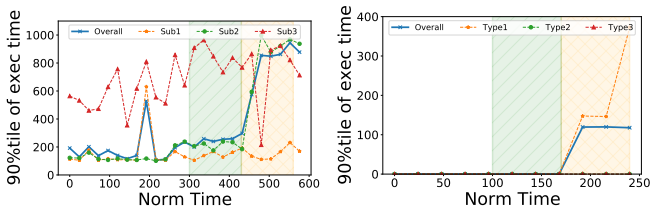

Fig. 4. Examples for diagnosing live migration performance issues – Case 1 (left) and Case 2 (right). The green and orange rectangles represent the background class and target class, respectively.

Incident-2 For another incident, CONAN found that the pattern with the highest objective function value contained a specific VM type (a.k.a SKU), i.e., {Component="B", Type="D2_V2"}, which means that this specific VM type had a sudden increase in Component B's execution time. After some investigations, it was discovered that a code bug in the VM agent, associated with the detected VM type, resulted in unexpected lock contention. We show the daily 90th percentile of component B's execution time for all live migration processes (the solid line) as well as for 3 specific VM Types (the dashed lines) in Fig. 4 (right), respectively.

C. Diagnosis for Substrate

a) Background: Substrate is a large-scale management service at Microsoft 365 that runs on hundreds of thousands of machines globally. Substrate hosts data for services like Exchange and SharePoint. Two categories of telemetry data are collected. The first category is the request data accessing Substrate management APIs. There are several roles in Substrate such as FrontDoor (FD), Café, BackEnd (BE), etc. Café is a stateless front end that handles client connections and routes requests to backend servers. A request goes through a path that consists of several roles, e.g., FD $\Rightarrow$ Café $\Rightarrow$ BE. The end-to-end latency and more fine-grained latencies of each role or between different checkpoints for each request are monitored. The second category of data is the telemetry data collected from the underlying machines, i.e., the CPU and memory utilization of critical service functions.

Service teams configure alerting rules for the latency and availability metrics. Any abnormal upward trend in latency or a significant drop in availability will trigger an alert. In practice, one major cause is code regression or configuration changes. As Substrate is made up of many components, it is challenging to quickly identify the actual cause.

b) Existing Practice: Once an alert is fired, engineers manually try various data pivots to check if the detected issue is more concentrated in a narrower scope, e.g., a specific datacenter or process. After that, they could dig deeper into the code to find out the actual root cause. However, due to the sheer volume of data being processed, visualizing a data pivot can take over 20 seconds, not to mention the huge number of attribute combinations.

c) Applying CONAN for latency issues: Each request is regarded as an instance. The contrast classes and the objective

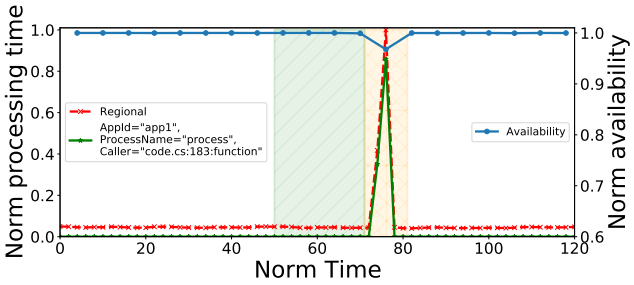

Fig. 5. The correlation between availability drop and regional CPU processing time surge. The processing time associated with the pattern was the dominating factor. The green and orange rectangles represent the background class and target class, respectively.

function are defined the same way as in Sec. V-B. The goal is to discover the pivot, i.e., a pattern, where the requests contribute most significantly to the latency increase.

Example: One day a latency spike is detected for a specific API. After several hours, engineers find that most of the prolonged requests came from a region S, but could not further narrow down the scope. By running CONAN, we obtain a pattern of {region="S", type="Customer"}. The interesting thing is that, according to the design, the API should skip the "Customer" type of users and return a constant immediately. However, after checking the code, the engineer found that a recent change accidentally modified the logic. The engineer admitted that it would take a long time to manually figure out this problem as it was due to unexpected behavior. Finally, a pull request was submitted to fix this issue, after which the latency increase disappeared.

d) Applying CONAN for availability issues: Many availability issues are caused by high CPU usage. To find out why the CPU usage is unexpectedly high, we apply CONAN by setting up the background and target classes as CPU processing time records before and during the anomaly period, respectively. Each CPU processing time record has the ClusterId, MachineId, AppId, ProcessName, Caller, etc. The objective function is defined as Eq. (5), which is the difference in the sum of CPU processing time values in the two classes.

$$f_{B,T}(p) = \sum CPU_T(p) - \sum CPU_B(p) \tag{5}$$

Example: In one past period of time, engineers observed that the availability of a specific API dropped occasionally. The root cause was difficult to pinpoint as it was not a fail-stop issue where the problem persists for a prolonged period of time. After some investigation, the engineer suspected that high CPU utilization was the cause as the drops in availability correlated well with surges in CPU processing time. Therefore, he applied CONAN to localize the possible cause by analyzing CPU processing time data. CONAN found a pattern of {AppId="app1", ProcessName="process", Caller="code.cs:183:function"}, with details shown in Fig. 5. After further investigation, the engineer realized that a recent configuration change led to an expensive CPU cost increase for a function in "code.cs". Interestingly, the cost increase was only triggered by a special workload, and therefore it only occurred from time to time and was hard to diagnose.

D. VM Unexpected Reboot

a) Background: VM unexpected reboot is one of the most common incident types in large-scale clouds like Azure. Like other modern cloud providers [51], [2], Azure adopts the storage-computing decoupling architecture. When a VM accesses its disks, it is unaware that the disks are remotely mounted, called virtual hard disks. One major category of VM unexpected reboots is due to disk disconnection. Disk disconnection could be caused by a variety of reasons such as a misconfiguration on the host OS, a failure on the network path connecting the VM and its disks, or a capacity throttling in the storage cluster.

In Azure, the Storage team constantly monitors the number of VM reboots caused by disk disconnection and needs to handle the incident when an anomalous spike is detected. We analyzed a 6-month period's incidents processed by the Storage team. Only $\sim 10\%$ were caused by storage issues. Therefore, it becomes a pain point for the Storage team, and they want to quickly transfer incidents to other teams if the root cause is not in storage.

b) Existing Practice: Each VM reboot caused by disk disconnection generates a record with main attributes such as NodeId, customer subscription Id, Compute/Storage cluster, Datacenter, network routers, etc. The storage team monitors the number of VM reboots for each datacenter. When an incident is detected, a triage workflow is triggered which aims to quickly localize the failure. The existing practice is based on one simple rule – if all failures are within one Storage cluster while distributed across multiple Compute clusters, it indicates a Storage issue. Similarly, they conclude it is a compute issue if all failures happen within a Compute cluster. This simple rule works well when the required conditions are met. However, it lacks robustness against ambient noise. In fact, random VM reboots are ambient in the cloud, and therefore, the perfect one-to-many relationship rarely exists. The simple rule can only cover less than 3% of incidents in the 6-month period. As a result, the engineer has to manually examine the logs for all remaining ones.

c) Applying CONAN: We treat each VM reboot event as one instance. The background class and target class are the VM reboot events that occur before and after the incident start time, respectively. The reboot events before the incident represent the ambient noise. The objective function is defined in Eq. (6). $S_T(p)$ and $S_B(p)$ are the sets of instances containing pattern p in the target class and background class, respectively. This objective function is used to capture the situation that the number of VM reboot events associated with pattern p has a significant emerging trend.

$$f_{B,T}(p) = |S_T(p)| - |S_B(p)| \tag{6}$$

The attribute hierarchy chains in this scenario are:

TABLE III
OUTPUT PATTERNS FOR DIAGNOSING VM REBOOT INCIDENTS

ID	Pattern	Evidence	Team
1	{TOR="tor1"}	Single TOR failure	PhyNet
2	{NodeId="n1"}	Single node failure	HostNet
3	{Storage="s1"}	Failure in a storage cluster	Storage
4	{Compute="c1"}	Failure in a compute cluster	Compute
5	{Sub = "sub1", Compute="cc1"}	Failure due to a subscription in a compute cluster	Compute
6	{Sub = "sub1", Storage="sc1"}	Failure due to a subscription in a storage cluster	Storage
7	{DC="dc1"}	Failure across datacenters	Unknown

- NodeId $\longrightarrow$ Storage Cluster $\longrightarrow$ Datacenter
- NodeId $\longrightarrow$ TOR $\longrightarrow$ Compute Cluster $\longrightarrow$ Datacenter

d) Results: We evaluated CONAN with a total of 51 historical incidents with groundtruth from their postmortems. CONAN could precisely triage 46 out of 51 incidents ($\sim 90\%$) to the responsible team as summarized in Table III. Among the incidents, 31 cases (*pattern 1*) were due to a single TOR failure, which should be recovered by the Physical Networking team. Eight cases (*pattern 2*) were single-node network issues handled by the Host Networking team (e.g., cable disconnection, attack, etc.). Three cases (*pattern 4, 5*) were caused by maintenance issues or abnormal subscription behaviors in a compute cluster, and another three cases (*pattern 3, 6*) due to extreme traffic load for a certain subscription in one storage cluster, which should be processed by the Storage team. For the remaining 6 cases, CONAN reports a pattern with a single datacenter (*pattern 7*), in which 4 were caused by a region-level DNS issue and 1 by a flawed configured virtual network. Though CONAN was unable to anticipate the responsible team for these cases, *pattern 7* implies large-scale failures across datacenters which usually require collaborative investigations from multiple teams. CONAN takes only tens of milliseconds for each VM reboot incident, which is negligible.

E. Node Fault

a) Background: Virtual machines are hosted on physical computing nodes. Node faults have various root causes such as hardware faults, OS crashes, host agent failures, network issues, etc. Due to the high complexity of cloud systems, quick identification of the failure category has become a key challenge for operators and developers.

Monitors are set up to detect the upsurge in the number of nodes reporting the same fault code. Each fault code represents a certain coarse-grained failure category, such as network timeout, failed resource access, permission issue, etc. For further digging out the root cause of triggered alerts, console logs on each node are collected from various sources such as OS, hardware, host agent, network, etc. Each log contains the node ID, a timestamp, and a severity level such as INFO, WARNING, EXCEPT or ERROR, and raw messages. However, due to the verbose details and large-scale data volume, the clues of the current issue are usually buried in massive logs, and thus, it can take long to locate logs that are indicative of the root cause of a node fault incident.

b) Existing Practice: The current process of exploring the logs is ad-hoc and inefficient. Once a node fault incident is fired, the on-call engineers manually sample a set of faulty nodes. Then, they usually pull some logs with severe verbosity levels (such as EXCEPT and ERROR) and examine these logs line by line or search for keywords based on their domain expertise.

c) Applying CONAN: Each node is regarded as one instance. We set up the background and target classes as healthy nodes and nodes reporting the fault code during the incident period, respectively. Both healthy nodes and faulty nodes are from the same cluster.

We intended to discover a set of logs that are more prevalent in the target class (i.e., nodes reporting the fault code) than the background class (i.e., healthy nodes). We thus used a similar objective function as defined in Eq. (2). However, console log messages are text-like data, which are not categorical variables as in previous scenarios. A console log is typically composed of a constant template and one or more variables. For instance, the log message "*Node[001], Received completion, Status:0x0*" contains two variables underlined. To compare different log sets, we remove the variables using a log parser [45]. For example, the aforementioned log message is converted into a template of "*Node[*], Received completion, Status: **". Only the constant tokens are retained and the changing variables are replaced with '*'s. All logs with the same template are assigned the same log Id. CONAN is applied to identify a set of log Ids to differentiate the faulty and healthy nodes.

d) Example: We encountered an incident about the timeout of a critical workflow on a group of nodes. This workflow involves multiple production teams. Any incorrect operation, such as a bad deployed package and certificate expiration, could trigger the timeouts. Logs were collected from the target and background nodes, respectively. CONAN was used to find a group of suspicious logs to report to the engineer. Soon, the engineer inferred that the incident was caused by a certificate file missing as CONAN reported one INFO log "Remove certificate for Service:host.pfx". It turned out that the certificate was removed from service resources by mistake. One interesting lesson learned from this case is that the most critical logs are not necessary EXCEPTs or ERRORs.

VI. PRACTICAL EXPERIENCES

a) Deployment model: We provide two ways to integrate CONAN into diagnostic tools. The first is to upload data to our service and consume the report in an asynchronous fashion. The second is to integrate our Python lib into an existing pipeline. The common practice is that a service team (owner for diagnosing certain batch failures) first evaluates CONAN using a few cases with our web service. After confirming the effectiveness of CONAN, the service team integrates our Python lib, which is more customizable and efficient, into their automation pipeline that is triggered by incidents.

b) The batch size: Cloud incidents often involve tens or thousands of instances, e.g., nodes, VMs, or requests. A

failure of a single or a few instances would not be escalated to an incident because of fault-tolerant mechanisms. Admittedly, a minimal number of instances are needed to guarantee the statistical significance of the objective function. In CONAN, we include the contrast patterns together with the number of instances involved in the output. The user can thus understand the significance of the patterns.

c) Continuous attribute values: In some scenarios, the attribute value is continuous rather than categorical. Data discretization has been extensively studied in the literature [30], [36], which could convert the raw continuous value into discrete buckets. The discretization could be also based on domain knowledge. For instance, in one of our real-world scenarios, the file size is a continuous value that typically ranges from less than 1 KB to several GB. According to expert knowledge, the file size is categorized into tiny ($<$ 1K), small (1K-64K), median (64K-1M), large (1M-10M), and ex-large ($>$10M).

d) Handling temporal patterns: Performance metrics usually exhibit temporal patterns. For example, the response latency varies in a peak/valley pattern during the day/night period, respectively. When we need to contrast two sets of data, we need to incorporate the temporal pattern. For instance, when the target class is during the peak hours, we should set up the background class also from peak hours for a fair comparison.

e) Relationship with A/B testing: A/B testing is a widely used technique for testing. The key idea of A/B testing is to proactively control the environment so that typically only one variable has two versions in the experiments. In this work, we also have two classes of data, i.e., target and background, which however are based on passive observations. Because the environment is not controlled, we do not know which variable we should test. Moreover, the root cause could be a combination of multiple variables. Thus, we use a combinatorial search algorithm to find the contrast pattern.

VII. Threats to Validity

The threats to validity mainly come from the early problem definition stage when we need to select the input data for diagnosis, set up the contrast classes, and specify the right objective function as discussed. Misconfigurations in practice may hinder the effectiveness of CONAN. For instance, the input data may be wrongly aggregated leading to important information missing; root-cause-indicating attributes may be excluded from our analysis or the objective function may be inappropriately configured, leading to failure in finding significant contrast patterns. In conclusion, the successful application of CONAN relies on engineer experiences to correctly formulate the diagnosis problem as discussed in Sec. III-B. To reduce the threats, we introduce several example scenarios as the best practices, so that the users can avoid common mistakes when applying CONAN to diagnose their new batch failure problems. In this paper, we carefully selected and detailed the application processes of CONAN in a systematic fashion for all typical scenarios. These scenarios are diverse types of instances, methodologies for setting up contrast classes, objective functions for availability, and performance problems. We have followed this practice to onboard several new diagnosis scenarios in an effective way.

VIII. Related Work

Diagnosis in cloud systems is a broad topic that has been studied extensively in the literature. Due to the hardware/software complexity of the modern cloud, it is critical to understand its behaviors by using the end-to-end tracing infrastructure. Among those works, Ding et al. [50] proposed using a proactive logging mechanism to enhance failure diagnosis. X-ray [13] compared different application control flows to identify the root cause of performance issues. Canopy [29] showed the benefit of end-to-end tracing to assist human inference. Dan et al. [9] collected metrics from controlled experiments in a live serving system to diagnose performance issues. In our work, CONAN also leverages the tracing infrastructure in Azure and Microsoft 365 to collect contextual information for batch failure diagnosis.

Batch failure is a common type of failure in the cloud and large-scale online services, which has attracted substantial attention as summarized in Table I. Although inferring diagnosis hints by setting two contrast categories is not new, this work is the first time to provide a generic framework in industrial scenarios. There is also some diagnosis work [23], [28], [10], [37] that leverages the service dependency graph or invocation chain to localize the root cause. Other work [27], [11], [24] proposes to use agents in distributed systems for failure detection and localization.

Contrast pattern [15], [22] has been proposed over two decades. Because of its differentiation nature, the idea of contrast pattern analysis has been applied to various diagnosis problems. Yu et al. [49] use contrast analysis for identifying the root cause of slow Window event traces. FDA [33], Sweeper [38], CCSM [41], and Marco et al. [19] apply association rule mining to find suspicious behavior in software crash reports. The primary drawback of existing work is that they usually target a very specific scenario, and thus, hard to be extended to similar situations. Besides, extracting contrast patterns is a computationally challenging problem. CONAN adopts a meta-heuristic search framework inspired by combinatorial test generation [34].

IX. Conclusion

In this work, we propose CONAN, a framework to assist the diagnosis of batch failures, which are prevalent in large-scale cloud systems. At its core, CONAN employs a meta-heuristic search algorithm to find contrast patterns from the contextual data, which is represented by a unified data model summarized from various diagnosis scenarios. We introduce our practices in successfully integrating CONAN into multiple real products. We also compare our search algorithm with existing approaches to show its efficiency and effectiveness.

REFERENCES

[1] Amazon AWS Power Outage. https://www.bleepingcomputer.com/news/technology/amazon-aws-outage-shows-data-in-the-cloud.-is-not-always-safe/. [Online; accessed 08-July-2022].

[2] Amazon EC2 Root Device Volume. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/RootDeviceStorage.html. [Online; accessed 08-May-2022].

[3] Azure VM Live Migration. https://azure.microsoft.com/en-us/blog/improving-azure-virtual-machine-resiliency-with-predictive-ml.-and-live-migration/. [Online; accessed 10-Sep-2022].

[4] Performance Buckets in Application Insights. https://azure.microsoft.com/en-us/blog/performance-buckets-in-application-insights/l. [Online; accessed 08-May-2022].

[5] Safe deployment practices. https://docs.microsoft.com/en-us/devops/operate/safe-deployment-practices. [Online; accessed 03-Sep-2022].

[6] Google Cloud Status Dashboard. https://status.cloud.google.com/summary, 2021. [Online; accessed 28-Apr-2021].

[7] AWS Post-Event Summaries. https://aws.amazon.com/cn/premiumsupport/technology/pes/, 2022. [Online; accessed 28-Apr-2022].

[8] Azure status history. https://status.azure.com/en-us/status/history/, 2022. [Online; accessed 28-Apr-2022].

[9] Dan Ardelean, Amer Diwan, and Chandra Erdman. Performance analysis of cloud applications. In *NSDI*, 2018.

[10] Behnaz Arzani, Selim Ciraci, Luiz Chamon, Yibo Zhu, Hongqiang (Harry) Liu, Jitu Padhye, Boon Thau Loo, and Geoff Outhred. 007: Democratically finding the cause of packet drops. In *NSDI*, 2018.

[11] Behnaz Arzani, Selim Ciraci, Boon Thau Loo, Assaf Schuster, and Geoff Outhred. Taking the blame game out of data centers operations with netpoirot. In *SIGCOMM*, 2016.

[12] Emre Ates, Lily Sturmann, Mert Toslali, Orran Krieger, Richard Megginson, Ayse K. Coskun, and Raja R. Sambasivan. An Automated, Cross-Layer Instrumentation Framework for Diagnosing Performance Problems in Distributed Applications. In *SoCC*, 2019.

[13] Mona Attariyan, Michael Chow, and Jason Flinn. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *OSDI*, 2012.

[14] Chetan Bansal, Sundararajan Renganathan, Ashima Asudani, Olivier Midy, and Mathru Janakiraman. Decaf: Diagnosing and triaging performance issues in large-scale cloud services. In *ICSE*, 2020.

[15] Stephen D. Bay and Michael J. Pazzani. Detecting change in categorical data: Mining contrast sets. In *KDD*, 1999.

[16] Ranjita Bhagwan, Rahul Kumar, Chandra Sekhar Maddila, and Adithya Abraham Philip. Orca: differential bug localization in large-scale services. In *OSDI*, 2018.

[17] Christian Blum and Andrea Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Comput. Surv.*, 35(3), 2003.

[18] Shaowei Cai. Balance between complexity and quality: Local search for minimum vertex cover in massive graphs. In *IJCAI*, 2015.

[19] Marco Castelluccio, Carlo Sansone, Luisa Verdoliva, and Giovanni Poggi. Automatically analyzing groups of crashes for finding correlations. In *ESEC/FSE*, 2017.

[20] Zhuangbin Chen, Yu Kang, Liqun Li, Xu Zhang, Hongyu Zhang, Hui Xu, Yangfan Zhou, Li Yang, Jeffrey Sun, Zhangwei Xu, Yingnong Dang, Feng Gao, Pu Zhao, Bo Qiao, Qingwei Lin, Dongmei Zhang, and Michael R. Lyu. Towards intelligent incident management: Why we need it and how we make it. In *ESEC/FSE*, 2020.

[21] Bhaskar Dasgupta, Xin He, Tao Jiang, Ming Li, John Tromp, Lusheng Wang, and Louxin Zhang. Handbook of combinatorial optimization, 1998.

[22] Guozhu Dong and Jinyan Li. Efficient mining of emerging patterns: Discovering trends and differences. In *KDD*, 1999.

[23] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices. In *ASPLOS*, 2019.

[24] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, Zhi Wei Lin, and Varugis Kurien. Pingmesh: A Large-scale system for data center network latency measurement and analysis. In *SIGCOMM*, 2015.

[25] Shilin He, Qingwei Lin, Jian-Guang Lou, Hongyu Zhang, Michael R Lyu, and Dongmei Zhang. Identifying impactful service system problems via log analysis. In *ESEC/FSE*, 2018.

[26] Holger Hoos and Thomas Sttzle. *Stochastic Local Search: Foundations & Applications*. Morgan Kaufmann Publishers Inc., 2004.

[27] Peng Huang, Jacob R Lorch, Lidong Zhou, Chuanxiong Guo, and Yingnong Dang. Capturing and Enhancing In Situ System Observability for Failure Detection Capturing and Enhancing In Situ System Observabil. In *OSDI*, 2018.

[28] Hiranya Jayathilaka, Chandra Krintz, and Rich Wolski. Performance monitoring and root cause analysis for cloud-hosted web applications. In *WWW*, 2017.

[29] Jonathan Kaldor, Jonathan Mace, Michał Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, Vinod Venkataraman, Kaushik Veeraraghavan, and Yee Jiun Song. Canopy: An end-to-end performance tracing and analysis system. In *SOSP*, 2017.

[30] Randy Kerber. Chimerge: Discretization of numeric attributes. In *AAAI*, 1992.

[31] Sebastien Levy, Randolph Yao, Youjiang Wu, Yingnong Dang, Peng Huang, Zheng Mu, Pu Zhao, Tarun Ramani, Naga Govindaraju, Xukun Li, et al. Narya: Predictive and adaptive failure mitigation to avert production cloud vm interruptions. In *OSDI*, 2020.

[32] Ze Li, Qian Cheng, Ken Hsieh, Yingnong Dang, Peng Huang, Pankaj Singh, Xinsheng Yang, Qingwei Lin, Youjiang Wu, Sebastien Levy, and Murali Chintalapati. Gandalf: An intelligent, end-to-end analytics service for safe deployment in large-scale cloud infrastructure. In *NSDI*, 2020.

[33] Fred Lin, Keyur Muzumdar, Nikolay Pavlovich Laptev, Mihai-Valentin Curelea, Seunghak Lee, and Sriram Sankar. Fast dimensional analysis for root cause investigation in a large-scale service environment. *SIGMETRICS Perform. Eval. Rev.*, 2020.

[34] Jinkun Lin, Chuan Luo, Shaowei Cai, Kaile Su, Dan Hao, and Lu Zhang. TCA: An Efficient Two-Mode Meta-Heuristic Algorithm for Combinatorial Test Generation. In *ASE*, 2015.

[35] Qingwei Lin, Jian-Guang Lou, Hongyu Zhang, and Dongmei Zhang. iDice: Problem identification for emerging issues. In *ICSE*, 2016.

[36] Huan Liu, Farhad Hussain, Chew Lim Tan, and Manoranjan Dash. Discretization: An enabling technique. *Data Min. Knowl. Discov.*, 2002.

[37] Meng Ma, Jingmin Xu, Yuan Wang, Pengfei Chen, Zonghua Zhang, and Ping Wang. Automap: Diagnose your microservice-based web applications automatically. In *WWW*, 2020.

[38] Vijayaraghavan Murali, Edward Yao, Umang Mathur, and Satish Chandra. Scalable statistical root cause analysis on app telemetry. ICSE-SEIP, 2021.

[39] Karthik Nagaraj, Charles Killian, and Jennifer Neville. Structured comparative analysis of systems logs to diagnose performance problems. In *NSDI*, 2012.

[40] Jamie Pool, Ebrahim Beyrami, Vishak Gopal, Ashkan Aazami, Jayant Gupchup, Jeff Rowland, Binlong Li, Pritesh Kanani, Ross Cutler, and Johannes Gehrke. Lumos: A library for diagnosing metric regressions in web-scale applications. In *KDD*, 2020.

[41] Rebecca Qian, Yang Yu, Wonhee Park, Vijayaraghavan Murali, Stephen Fink, and Satish Chandra. Debugging crashes using continuous contrast set mining. In *ICSE-SEIP*, 2020.

[42] Raja R. Sambasivan, Alice X. Zheng, Michael De Rosa, Elie Krevat, Spencer Whitman, Michael Stroucken, William Wang, Lianghong Xu, and Gregory R. Ganger. Diagnosing performance changes by comparing request flows. In *NSDI*, 2011.

[43] Cheng Tan, Ze Jin, Chuanxiong Guo, Tianrong Zhang, Haitao Wu, Karl Deng, Dongming Bi, and Dong Xiang. Netbouncer: Active device and link failure localization in data center networks. In *NSDI*, 2019.

[44] Cheng Tan, Ze Jin, Chuanxiong Guo, Tianrong Zhang, Haitao Wu, Karl Deng, Dongming Bi, Dong Xiang, and Implementation Nsdi. NetBouncer: Active Device and Link Failure Localization in Data Center Networks. In *NSDI*, pages 1–14, 2019.

[45] Xuheng Wang, Xu Zhang, Liqun Li, Shilin He, Hongyu Zhang, Yudong Liu, Lingling Zheng, Yu Kang, Qingwei Lin, Yingnong Dang, Saravan Rajmohan, and Dongmei Zhang. SPINE: A Scalable Log Parser with Feedback Guidance. In *FSE*, 2022.

[46] Yiyuan Wang, Shaowei Cai, and Minghao Yin. Local search for minimum weight dominating set with two-level configuration checking and frequency based scoring function. In *IJCAI*, 2017.

[47] Rongxin Wu, Hongyu Zhang, Shing-Chi Cheung, and Sunghun Kim. Crashlocator: Locating crashing faults based on crash stacks. In *ISSTA*, 2014.

[48] Tianyin Xu, Xinxin Jin, Peng Huang, Yuanyuan Zhou, Shan Lu, Long Jin, and Shankar Pasupathy. Early Detection of Configuration Errors to Reduce Failure Damage. In *OSDI*, 2016.

[49] Xiao Yu, Shi Han, Dongmei Zhang, and Tao Xie. Comprehending performance from real-world execution traces: A device-driver case. In *ASPLOS*, 2014.

[50] Ding Yuan, Soyeon Park, Peng Huang, Yang Liu, Michael M. Lee, Xiaoming Tang, Yuanyuan Zhou, and Stefan Savage. Be conservative: Enhancing failure diagnosis with proactive logging. In *OSDI*, 2012.

[51] Qiao Zhang, Guo Yu, Chuanxiong Guo, Yingnong Dang, Nick Swanson, Xinsheng Yang, Randolph Yao, Murali Chintalapati, Arvind Krishnamurthy, and Thomas Anderson. Deepview: Virtual Disk Failure Diagnosis and Pattern Detection for Azure. In *NSDI*, 2018.

[52] Xu Zhang, Chao Du, Yifan Li, Yong Xu, Hongyu Zhang, Si Qin, Ze Li, Qingwei Lin, Yingnong Dang, Andrew Zhou, et al. HALO: Hierarchy-aware Fault Localization for Cloud Systems. In *SIGKDD*, 2021.

[53] Xu Zhang, Yong Xu, Si Qin, Shilin He, Bo Qiao, Ze Li, Hongyu Zhang, Xukun Li, Yingnong Dang, Qingwei Lin, Murali Chintalapati, Saravanakumar Rajmohan, and Dongmei Zhang. Onion: Identifying incident-indicating logs for cloud systems. In *ESEC/FSE*, 2021.

[54] Nengwen Zhao, Junjie Chen, Zhou Wang, Xiao Peng, Gang Wang, Yong Wu, Fang Zhou, Zhen Feng, Xiaohui Nie, Wenchi Zhang, Kaixin Sui, and Dan Pei. Real-time incident prediction for online service systems. In *ESEC/FSE*, 2020.

[55] Günther Zäpfel, Roland Braune, and Michael Bögl. *Metaheuristic search concepts: A tutorial with applications to production and logistics*. 2010.