# FluxRank: A Widely-Deployable Framework to Automatically Localizing Root Cause Machines for Software Service Failure Mitigation

Ping Liu †¶, Yu Chen§, Xiaohui Nie†¶, Jing Zhu†¶,
Shenglin Zhang**, Kaixin Sui‡, Ming Zhang ‖, Dan Pei*†¶
†Tsinghua University §Baidu **Nankai University ‡BizSeer ‖China Construction Bank
¶Beijing National Research Center for Information Science and Technology (BNRist)

***Abstract*—The failures of software service directly affect user experiences and service revenue. Thus operators monitor both service-level KPIs (*e.g.*, response time) and machine-level KPIs (*e.g.*, CPU usage) on each machine underlying the service. When a service fails, the operators must localize the root cause machines, and mitigate the failure as quickly as possible. Existing approaches have limited application due to the difficulty to obtain the required additional measurement data. As a result, failure localization is largely manual and very time-consuming.**

**This paper presents FluxRank, a widely-deployable framework that can automatically and accurately localize the root cause machines, so that some actions can be triggered to mitigate the service failure. Our evaluation using historical cases from five real services (with tens of thousands of machines) of a top search company shows that the root cause machines are ranked top 1 (top 3) for 55 (66) cases out of 70 cases. Comparing to existing approaches, FluxRank cuts the localization time by more than 80% on average. FluxRank has been deployed online at one Internet service and six banking services for three months, and correctly localized the root cause machines as the top 1 for 55 cases out of 59 cases.**

***Index Terms*—KPI, failure mitigation, recommendation**

## I. Introduction

The failures of a software service directly affect user experiences and service revenue [1–3]. Thus service operators monitor both service-level KPIs (Key Performance Indicators) (*e.g.*, response time) and machine-level KPIs (*e.g.*, CPU usage) on each machine (*e.g.*, servers or virtual machines) underlying the service [4, 5]. When some service KPIs become anomalous, indicating a service failure, the operators must mitigate the failure as quickly as possible.

In practice, the process of troubleshooting commonly consists of three steps: failure confirmation, mitigation and root cause analysis. As shown in Fig. 1, when a critical KPI (*e.g.*, response time) of a software service is anomalous, on-call operators first take a few minutes to confirm if the service really has a failure. If the failure is confirmed, then operators must immediately take some actions (*e.g.*, switch traffic away from the faulty machines or restarting the faulty machines) to put the service back to normal as soon as possible, without pinpointing the *exact* root cause (*e.g.*, the exact reason *why* the machines are faulty). The mitigation time must be as short in order to prevent larger loss. After a successful mitigation, *developers* have enough time to find and fix the exact root cause (*e.g.*, bugs in the code, configuration or design).

It is important to highlight that a mitigation process is different from a root cause analysis process. A large software

*Dan Pei is the correspondence author.

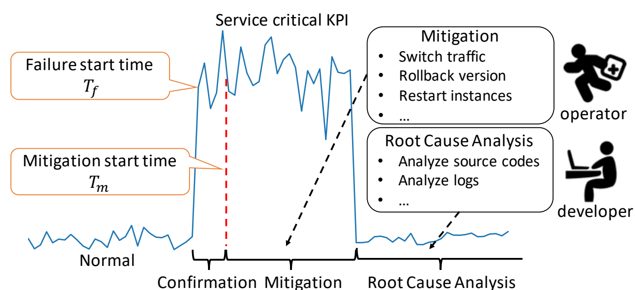


Fig. 1: The process of troubleshooting. When a critical KPI (*e.g.*, response time) of a software service is anomalous, on-call operators first take a few minutes to confirm if the service actually has a failure. If the failure is confirmed, then operators must immediately take mitigation actions to return the service to normal. After a successful mitigation, developers have enough time to localize the root cause by analyzing the source codes, logs, configurations, *etc.*.

service is typically a complex distributed system, consisting of many (*e.g.*, 10∼100) modules (*e.g.* web server module, database module, computation module, *etc.*). Each module could be deployed on many (*e.g.* hundreds of) machines in multiple data centers, and each machine can have more than 100 machine-level KPIs. Thus, it is both challenging and often unnecessary for operators to quickly pinpoint the *exact* root cause before the mitigation, for the following two reasons. On one hand, at the time of mitigation, operators typically do not have the necessary details for the root cause (design, codes, application logs *etc.*) of the service. On the other hand, localizing the faulty machines, indicated by changes of machine-level KPIs, is often sufficient to mitigate the failure in modern service architecture (*e.g.* microservice) using tools such as Kubernetes [6] and Mesos [7].

### A. Limitations of Existing Approaches

In this paper, we focus on *localizing* the root causes to the extent (*e.g.* faulty machines) where mitigation can be done (*e.g.* rebooting) as opposed to *analyzing* the exact root cause. There are many previous works on root cause localization. Most of these works [8–18] localize root cause by constructing dependency graphs. some works [13–18] focus on localizing root cause in computer *networks*, and the dependencies are inferred from the links in the network topology. However, a software service's dependency graph cannot be inferred by this way. Sherlock [8] needs to deploy an agent on each host

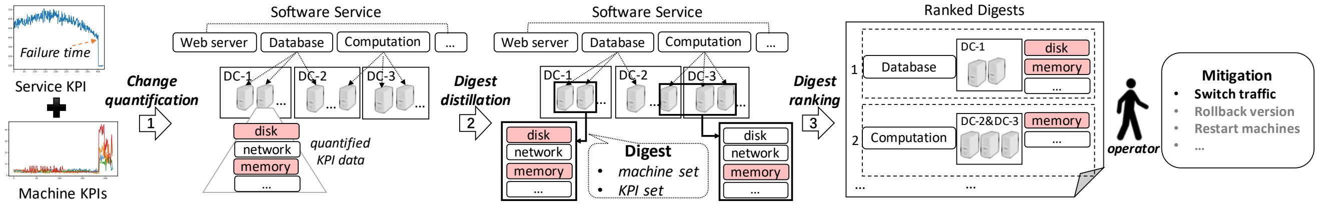


Fig. 2: Overview of FluxRank

to infer the dependency graph, then localizes the root cause by analyzing correlations across logs from the same clients and across different clients. MonitorRank [9] uses the trace logs to construct the dependency graph, and then localizes the root cause by a random walk algorithm. CauseInfer [10] uses wire-captured service calls to construct the service dependency graph and applies the PC-algorithm [19] to construct a metric causality graph, then localizes the root cause across the two level graphs. FChain [11] constructs the dependency graph by Sherlock [8]'s algorithm. BRCA [12] mines the dependency graph based on service's historical KPI anomaly alerts.

The dependency graphs required by these approaches are often very challenging to obtain in practice due to the difficulty to collect the additional necessary data mentioned above. Furthermore, it is also very challenging to maintain the graphs for the rapidly changing software services, especially those developed under the philosophy of Agile and DevOps. The quick change of the codes makes the dependency graph elusive. As a result, the application of the above approaches faces a lot of limitations, and the root cause localization in practice is still mainly manual and time-consuming.

### B. Intuitions and Core Ideas

Solving real world troubleshooting problems using machine learning is promising (a lot of monitoring data). However, directly training machine learning models using monitoring data in an end-to-end manner does not work due to: 1) not well defined problems, 2) incomplete information, 3) insufficient failure cases, and 4) the lack of interpretability. Our core high-level idea is to utilize domain knowledge (which is often not captured or learnable in the data, but abstractable or mimicable from manual troubleshooting process) to design a system architecture in which each component has a well defined problem, sufficient data, and an interpretable algorithm. This way, a seemingly unsolvable end-to-end machine learning problem in troubleshooting becomes solvable with a domain knowledge inspired architecture.

Therefore, our idea is inspired by the current manual five-step failure mitigation method developed over the years by a top global search engine service provider $S$ (that we work with). 1) In the background, operators use some online statistical algorithms (*e.g.*, static threshold) [20–22]) to detect anomalies on large number of service and machine KPIs. 2) After a service fails, operators *manually* scan through machine KPIs to find the set of KPIs and faulty machines which become anomalous *around the service failure time*. 3) Then operators *manually* rank the potential combinations of faulty machines according to operators' experience. 4) To mitigate the failure, operators *manually* trigger automatic action on the highest-ranked machine combinations one by one (*e.g.*, switch traffic away from the fault machines or restarting the faulty machines). 5) After a successful mitigation, the development engineers (not the operators) will take time to analyze the codes and logs to find and fix the exact root cause (*e.g.*. bugs in code, configuration.)

In the above approach, simply localizing the root cause machines and getting rid of them from the service can already mitigate the service failure, since there are typically many other working machines whose functions are identical to those of the faulty machines, taking advantage of the high-availability architecture prevalent in software services.

However, in the above approach, the first step is inefficient and inaccurate (see more details in Challenge 1 below), and the second and third steps are manual, thus the failure mitigation still take too long in $S$. We analyzed 8 recorded failure cases in a software service (with 29 modules running on 11,519 machines) during a 9-month period in $S$. The maximum mitigation time is 175 minutes and the mean is 59 minutes.

Inspired by above practical approach and realizing its inefficiency, this paper proposes a widely-deployable system called **FluxRank** that can automatically localize the root cause machines underlying a given software service failure. FluxRank uses the same measurement data, *i.e.* machine KPIs, readily available in almost all software services, and mimics the first three steps of the above approach, but makes each step efficient, accurate, and automatic so that the overall mitigation time can be greatly reduced.

Fig. 2 shows the overview of FluxRank. FluxRank is triggered by the service failure (*i.e.*, service KPI becomes anomalous. Anomaly detection of service KPI [20–29] is not part of FluxRank). The input of FluxRank includes: service failure time and all machine KPI data. It has three phases (see the bold arrows in Fig. 2): *change quantification*, *digest distillation* and *digest ranking*. First, the changes of all machine KPIs are quantified in the *change quantification* phase. Second, all KPIs are organized into *digests* by a clustering algorithm in the *digest distillation* phase. Each *digest* contains a set of machines that are from the same module and a set of KPIs that represent the anomalous pattern of the module. Third, all *digests* are automatically ranked by their potential as the root cause location in the *digest ranking* phase. Finally, based on the digest ranking results, operators mitigate the loss by triggering some automatic actions. *Change quantification* mimics step 2 in the manual mitigation process, while *digest distillation* and *digest ranking* mimics step 3.

### C. Challenges and Contributions

Below we summarize the challenges faced by **FluxRank** and this papers' contributions.

**Challenge 1: How to quickly quantify the changes of massive number of diverse KPIs around the service failure time?** Although many anomaly detection algorithms [20–27] have been proposed over the years, each monitored KPI needs its own algorithm and parameters. Given the large number (*e.g.*, millions) of diverse KPIs, careful algorithm selection and parameter tuning for each KPI becomes infeasible, resulting in inaccurate anomaly detection results and more difficulties in steps 2 and 3. However, different from a general anomaly detection scenario where the anomaly can happen at any time, in our scenario the service failure time is already given and we just need to develop an algorithm that can quantify the changes of machine KPIs around the service failure time. The remaining challenge is that the algorithm must be flexible to work on diverse KPIs, and be lightweight and fast enough so that it can quickly analyze the massive number of KPIs.

**Challenge 2: How to cluster KPIs with physical significance?** A failure in one module can propagate to other modules, and a faulty machine might have multiple KPIs anomalous at the same time. As a result, one service failure often coincides in time with many machine KPI anomalies. Operators often summarize the machine KPI anomalies according to their domain knowledge in an ad hoc manner. Our challenge is to automatically and systematically cluster the KPIs with similar anomaly degrees at the related machines/modules in a way that is intuitive to the operators to determine the mitigation actions.

**Challenge 3: How to rank the clustering results?** For a large software service, it is often the case that there are often concurrent but unrelated anomalies at the machine level. This means that, for a service failure, above phase 2 might output multiple clustering results. Therefore, in phase 3 we have to rank the clustering results from phase 2 such that the ones which are most relevant to the failure should appear at the top. However, ranking clusters (different KPIs with different anomaly degrees at different machines) have not been studied before, and there are no known metrics or ranking algorithm that can directly deal with such clustering results.

This paper's contributions can be summarized as follows:

**Contribution 1**. We propose a non-parametric lightweight algorithm based on Kernel Density Estimation for quantifying and comparing the changes of large number of diverse KPIs around a specific time, addressing challenge 1.

**Contribution 2**. To address challenge 2, we propose a vector representation of the changes, a distance function, and a DBSCAN-based clustering algorithm that organizes the KPIs into *digests* to represent the anomaly patterns of the modules.

**Contribution 3**. To address challenge 3, we propose a feature vector and a logistic regression based ranking algorithm that can rank the Root Cause Digest (RCD) at the top.

**Contribution 4**. Our experiments on 70 real offline failure cases of five production services (with tens of thousands of machines) in a top global software service provider show that the RCDs can be ranked to top one for 55 cases, and to top three for 64 cases. Compared to existing approach, FluxRank reduces the localization time by more than 80% on average.

**Contribution 5**. FluxRank is a widely-deloyable framework. We have successfully deployed FluxRank on one Internet services (with hundreds of machines) and six banking services (each with tens of machines). Over the course of three months of online deployment, it successfully analyzed 59 real cases, and the result shows the RCDs can be ranked to top one for 55 cases.

The rest of the paper is organized as follows. The design of FluxRank are presented in §II, §III, and §IV. §V introduces the system implementation. §VI presents our offline evaluation, and §VII presents operational experience in real deployment. Related work and conclusion are given in §VIII and §IX.

## II. Change Quantification

As aforementioned, in the phase of change quantification we try to quantify the changes of machine KPIs which are measured by *change degrees*. Change degrees can be used to compare among different types of KPIs (*e.g.*, CPU utilization, memory utilization, I/O rate). In addition, the KPIs of root cause machines will first change when a service failure happens, followed by the KPI changes of the machines that are impacted by this failure. Therefore, the *change start time ($T_c$)* is also helpful to localize root cause machines.

Clearly, the design goal of change quantification is to *rapidly and accurately identify the change start time and determine the change degree around the time of service failure for a large number of diverse KPIs.* Recall that traditional anomaly detection algorithms such as [20–23] cannot achieve the above goal because they are labor-intensive in algorithm selection and parameter tuning for a large number of diverse KPIs. Consequently, we propose to use a two-step design in change quantification: (1) apply absolute derivative to identify change start time and (2) use Kernel Density Estimation (KDE) to determine change degree.

### A. Change Start Time

Finding the change start time ($T_c$) of a KPI can be converted into a classic change point detection problem. Many previous methods, including supervised learning [30–34] and unsupervised learning [35–47], have been proposed to address this problem. For a large distributed software service, hundreds of thousands to millions of KPIs should be analyzed for a single service failure. Hence supervised learning methods cannot be used due to the infeasible labeling efforts. In addition, the algorithm should rapidly identify $T_c$ for mitigation, and thus the above unsupervised learning methods which are inefficient in computing cannot be applied in our scenario either.

As aforementioned, different from the general change point detection problem where a change can happen at any time, in our scenario the service failure time is given, and we just need to develop an algorithm that can detect KPI changes around the failure time. As shown in Fig. 1, the failure start time $T_f$ can be determined based on the service KPI, and the mitigation start time $T_m$ is when operators confirm the failure and start mitigation. Due to the delay of failure propagation, the KPIs of root cause machines can change before $T_f$. To identify the change start time of KPIs, we set a look-back window $[T_f - w_1, T_m]$, where $w_1$ is time length before $T_f$. In practice, $w_1$ is a configurable parameter. On the one hand, if $w_1$ is set too large, FluxRank may falsely include some KPI changes which are irrelative to the service failure. On the other

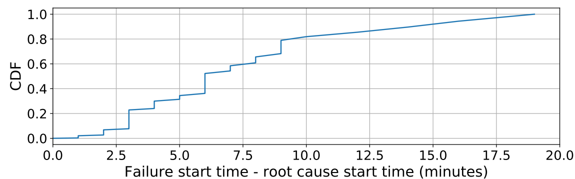

Fig. 3: The CDF of the delays between the change start time of root cause machines' KPIs and the start time of failures in 82 failures within one year in $S$.

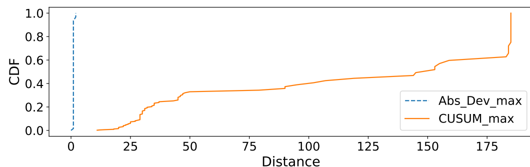

Fig. 4: Change point detection results of CUSUM [35] and Abs_Dev for 91 manually labeled anomalous KPIs. CUSUM_max and Abs_Dev_max are denoted the two algorithms that are applied the maximum output score.

hand, if $w_1$ is set too small, FluxRank may miss some KPI changes of root cause machines. After analyzing 82 failures within one year in $S$, we get the delays between the change start time of root cause machines' KPIs and the start time of failures as shown in Fig. 3, and find that 80% of the delays are less than 9 minutes, and the maximum delay is 19 minutes. Therefore, we empirically set $w_1$ to 30 min during evaluation.

In practice, the number of KPI data points within the look-back window can be small, because the monitoring interval can be per minute (50% of cases in our scenario), and the small number of data points cannot support the complex change point detection algorithms. We thus focus on the algorithms that involve only simple calculation. In this work, we desing a simple yet effective change point detection algorithm, Abs_Dev, which applys the absolute derivative of KPI data, to automatically, accurately and efficiently identify the change start time. Abs_Dev calculates the absolute derivative value [48] of each KPI data point. We deem that the KPI data point achieving the maximum output score (absolute derivative value) of Abs_Dev within the look-back window has the most significant change, and it represents the change start time. This way, the change start time is efficiently and accurately identified without any manual parameter tuning.

CUSUM [35] is a simple common change detection algorithm. To compare the performance of CUSUM [35] and Abs_Dev for change start time detection, we manually labeled the change start times ($T_c^{label}$) of 91 anomalous KPIs which contain all types of KPIs in our study. We use the distance between $T_c$ and $T_c^{label}$ to compare the performance of different algorithms: $distance = |T_c - T_c^{label}|$. Fig. 4 shows the detection results of CUSUM [35] and Abs_Dev. CUSUM_max and Abs_Dev_max are denoted the two algorithms that are applied the maximum output score method. We can see that the distances of Abs_Dev_max are much smaller than CUSUM_max. Further more, The distances of Abs_Dev_max are close to 0, which means the detected change start times of Abs_Dev_max are very close to the labeled correct change

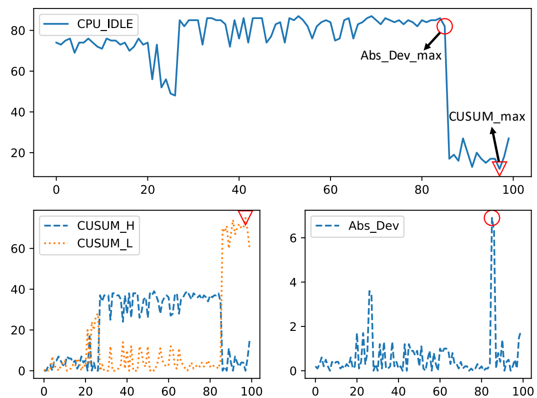

Fig. 5: The top figure is an anomalous CPU_IDLE KPI. The first change (in [20, 40]) is normal, and the second change (in [80, 100]) is anomalous. The bottom two figures are the output scores of CUSUM [35] and Abs_Dev when analyze the anomalous CPU_IDLE. CUSUM [35] has higher control CUSUM_H and lower control CUSUM_L. CUSUM_max denotes the maximum value of all data in CUSUM_H and CUSUM_L.

start times.

Specifically, Fig. 5 shows the two algorithms' output scores of an anomalous CPU_IDLE KPI which is one of the 91 anomalous KPIs. The first change ( in [20, 40]) of CPU_IDLE is normal, and the second change (in [80, 100]) is anomalous. Because derivative can directly reflect the change degree, we can see that the largest score of Abs_Dev corresponds to the correct change start time of the CPU_IDLE KPI. CUSUM [35] can also detect the correct change start time if the proper threshold can be chosen, but it is impossible to manually choose the thresholds for tens of thousands of machine KPIs. Thus we have to use CUSUM_max. Due to the cumulative sum, the score of CUSUM [35] will continue to increase if there are small changes after the correct change start time, as is the case in the bottom left of Fig. 5. So the largest score of CUSUM [35] cannot correspond to the correct change start time. Therefore, we choose Abs_Dev to detect the change start times of KPIs.

### *B. Change Degree*

As described in §I, the first challenge of FluxRank is to design a non-parametric lightweight algorithm for quantifying and comparing the changes of large number of diverse KPIs. Thus, we propose to use the observation probability of change to represent the change degree, because the probability is naturally quantitative and can be compared between different types of KPIs. We cannot find other metrics to represent and quantify KPI changes, which can satisfy the above requirements.

Specifically, the change start time $T_c$ is already obtained in §II-A, thus we collect data $\{x_i\}$ in the time interval $[T_c - w_2, T_c)$, and data $\{x_j\}$ in the time interval $[T_c, T_m]$ (we will introduce the selection of $w_2$ shortly). Obviously, $\{x_i\}$ are the data before the change and $\{x_j\}$ are the data after the change. Then, the mathematical representation of the observation probability of a change is: $P(\{x_j\}|\{x_i\})$. This formula denotes the probability of observing $\{x_j\}$ after change

TABLE I: The 47 types of machine KPIs

| Type (#number) | KPI & Corresponding Kernel Function |
|---|---|
| CPU-Related (8) | **Beta:** CPU_IDLE; CPU_HT_IDLE. **Poisson:**CPU_CONTEXT_SWITCH. **Gaussian:**CPU_INTERRUPT; CPU_SERVER_LOADAVG_1; CPU_SERVER_LOADAVG_15; CPU_SERVER_LOADAVG_5; CPU_WAIT_IO. |
| Disk-Related (15) | **Beta:** DISK_TOTAL_USED_PERCENT; FD_USED_PERCENT; DISK_TOTAL_INODE_USED_PERCENT. **Poisson:** DISK_FS_ERROR; FD_USED. **Gaussian:** DISK_PAGE_IN; DISK_PAGE_OUT; DISK_TOTAL_AVG_WAIT; DISK_TOTAL_IO_UTIL; DISK_TOTAL_READ_KB; DISK_TOTAL_READ_REQ; DISK_TOTAL_READ_AVG_WAIT; DISK_TOTAL_WRITE_AVG_WAIT; DISK_TOTAL_WRITE_KB; DISK_TOTAL_WRITE_REQ. |
| Memory-Related (6) | **Beta:** MEM_USED_PERCENT; MEM_USED_ADD_SHMEM_PERCENT. **Poisson:** MEM_BUFFERS; MEM_CACHED; MEM_USED; MEM_USED_ADD_SHMEM. |
| Network-Related (13) | **Beta:** NET_MAX_NIC_INOUT_PERCENT. **Poisson:** NET_TCP_IN_ERRS; NET_TCP_RETRANS; NET_TCP_LOSS; NET_UP_NIC_NUMBER. **Gaussian:** NET_TCP_ACTIVE_OPENS; NET_TCP_CURR_ESTAB; NET_TCP_IN_SEGS; NET_TCP_OUT_SEGS; NET_TCP_TIME_WAIT; NET_TOTAL_IN_BITPS; NET_TOTAL_OUT_BITPS; NET_TOTAL_SOCKETS_USED. |
| OS kernel-Related (5) | **Poisson:** SYS_OOM; SYS_PAGING_PROCS; SYS_RUNNING_PROCS; SYS_STOPPED_PROCS; SYS_ZOMBIE_PROCS. |

under the premise of observing $\{x_i\}$ before change. This probability indicates the degree of the change: the smaller the probability, the larger the change degree.

To calculate the probability of $P(\{x_j\}|\{x_i\})$, we first calculate the probability of each data point in $\{x_j\}$: $P(x_j|\{x_i\})$. Suppose that $\{x_i\}$ is generated by a random variable $X$, thus the probability distribution of $X$ can be estimated by $\{x_i\}$. To differentiate the upward changes and downward changes, we compute the overflow probability $P(X \geq x_j|\{x_i\})$ and the underflow probability $P(X \leq x_j|\{x_i\})$. We assume $\{x_j\}$ are independent identically distributed (i.i.d.) samples, then we can derive the overflow probability $P_o(\{x_j\}|\{x_i\})$ and underflow probability $P_u(\{x_j\}|\{x_i\})$ of $\{x_j\}$ as:

$$P_o(\{x_j\}|\{x_i\}) = \prod_{j=1}^{l} P(X \geq x_j|\{x_i\})$$

$$P_u(\{x_j\}|\{x_i\}) = \prod_{j=1}^{l} P(X \leq x_j|\{x_i\})$$

where $l$ is the number of data points in $\{x_j\}$. Apparently, a very small $P_o(\{x_j\}|\{x_i\})$ represents a upward change, and a very small $P_u(\{x_j\}|\{x_i\})$ denotes a downward change.

The number of samples in $\{x_j\}$ depends on not only the length of $(T_m - T_c]$, but also the collection interval of KPI data. If two KPIs have different collection intervals, $P_o(\{x_j\}|\{x_i\})$ and $P_u(\{x_j\}|\{x_i\})$ will be very different, because both $P_o(\{x_j\}|\{x_i\})$ and $P_u(\{x_j\}|\{x_i\})$ depend on the number of data points in $\{x_j\}$. In order to compare KPI changes with different collection intervals, we use the geometric mean directly, and convert the negative value to the positive value. Thus the scores of upward change $o$ and downward change $u$ would be:

$$o = -\frac{1}{l}\sum_{j=1}^{l} \log P(X \geq x_j|\{x_i\})$$

$$u = -\frac{1}{l}\sum_{j=1}^{l} \log P(X \leq x_j|\{x_i\})$$

Then, the remaining question is how to estimate the probability distribution of $X$ given $\{x_i\}$. A common solution is to assume that $X$ follows Gaussian distribution and estimate the probability distribution by sample mean and sample variance. However, many KPIs do not follow Gaussian distribution. For example, the KPI *CPU_IDLE*, which represents the percentage of CPU idle time, ranges from 0 to 1. It does not follow Gaussian distribution. So we adopt Kernel Density Estimation (KDE) [49], which can estimate proper probability distributions for different types of KPIs. KDE [49] is a non-parametric approach to estimate the probability density function (PDF) of a random variable based on observation samples.

$$\hat{f}(x) = \frac{1}{n}\sum_{i=1}^{n} K(x; x_i)$$

where $K$ is a kernel function which is determined by the physical meaning of $\{x_i\}$, and n is the size of $\{x_i\}$.

In the above framework, $\{x_i\}$ are used to estimate the probability distribution for random variable $X$. In order to obtain a precise model, $w_2$ should be large to more samples in $\{x_i\}$. However, many KPIs have periodic fluctuations, and the probability distribution of $X$ can change over time. Therefore, a too large $w_2$ will lead to inaccurate model, and we set $w_2 = 1h$ based on empirical experience.

After analyzing the physical meaning of 47 machine KPIs in our study, we find that three types of distributions can cover all the 47 KPIs: *Beta distribution*, *Poisson distribution* and *Gaussian distribution*, as shown in Table I. The same KPI's kernel function only needs to be selected once. Further, estimating distributions based on KPIs' physical meaning is more precise and stable than other sampling based methods which depend on the quality of the samples. The three distributions are discussed as follows.

*Beta Distribution:* it is the dual distribution of Binomial distribution. Binomial distribution describes the distribution of number of successes in a trial containing multiple Bernoulli tests, given that each Bernoulli test would result in a success with probability $p$. Beta distribution describes the probability distribution of $p$, given the number of tests and the number of successes. Therefore, it is suitable for the random behavior of percentages and proportions (*e.g.*, *CPU_IDLE*). For $x \in [0, 1]$, the PDF of Beta distribution is:

$$f(x; \alpha, \beta) = \frac{x^{\alpha-1}(1-x)^{\beta-1}}{B(\alpha, \beta)}$$

where $B$ is Beta function, and $\alpha$ and $\beta$ can be estimated by the *method of moments* [50], which is a commonly used estimation method for Beta distribution.

*Poisson Distribution:* it describes the number of times a random event occurs in a time interval or in a space. For example, *SYS_OOM*, which denotes the frequency of out of memory (OOM), follows Poisson distribution. The PDF of Poisson distribution is:

$$f(x; \lambda) = \frac{\lambda^x e^{-\lambda}}{x!}$$

where $\lambda$ is the expected number of events per interval, which can be set to $x_i$.

*Gaussian Distribution:* it is the default kernel function of KDE [49]. The KPIs will be modeled by Gaussian distribution if they cannot be modeled by the above two distributions. The details of Gaussian kernel function can be found in [49].

Finally, for each KPI, the change quantification algorithm outputs three values: upward change degree $o$, downward change degree $u$, and change start time $T_c$. $o$ and $u$ will be used for digest distillation, and $T_c$ will be used for digest ranking.

## III. Digest Distillation

For a large software service deployed in multiple data centers, it usually consists of 10$\sim$100 modules, and has tens of thousands of KPIs. Although the *Change Quantification* algorithm can filter out a lot of normal KPIs, there are still plenty of anomalous KPIs left. In one case of our experiment (§VI), the number of anomalous KPIs is 10,653. Manually Screening these KPIs one by one is still a daunting job for operators. To localize faulty machines quickly, these KPIs should *be organized in a legible way*. We propose to cluster the KPIs into *digests*, which is a legible way for operators to easily understand the status of the software service.

The basic idea of *Digest Distillation* is to cluster the machines with similar KPI change patterns. The input is all the quantified KPI changes in §II. First, the change degrees ($o$ and $u$) of each machine's KPIs can form a vector to represent the change pattern of the machine:

$$(o_0, u_0, ..., o_k, u_k)$$

where $o_k$ and $u_k$ are the change degrees of the KPI $k$. Second, the machines are clustered into digests (clusters) based on their vectors. We now introduce two key factors of clustering: distance function and clustering algorithm.

### A. Distance function

Distance function calculates the similarity between two vectors. Our intuition is to group into a cluster the machines whose KPIs change together. Although Euclidean distance is a widely adopted distance function, it cannot capture "two KPIs changing other" well. Euclidean distance measures the absolute distance of two vectors. When two KPIs change together, their Euclidean distance may be large for different change degrees but their correlation is very strong. Therefore, we adopt correlation as the distance function.

We compared three common correlation methods: *Pearson Correlation* [51], *Kendall's tau* [52], and *Spearman Correlation* [53]. The result $r$ of these three methods is a value in $[-1, 1]$, where 1 means a complete positive correlation, 0 means no correlation, and -1 means a complete negative correlation. We use the following formula to transform correlation $r$ into distance: $Distance = 1 - r$.

According to our experiment (§VI-C), *Pearson Correlation* [51] performs the best. The reason is that *Kendall's tau* [52] and *Spearman Correlation* [53] are ranking-based methods, thus little difference of the values will make the ranking change greatly. *Pearson Correlation* [51] is not based on the ranking, thus it is more suitable for our study.

### B. Clustering Algorithm

Currently, four classic clustering algorithms are commonly used: K-means [54], Gaussian mixture [55], hierarchical clustering [55], and DBSCAN [56].

K-means [54] and Gaussian mixture [55] both require the number of clusters as the input, which is not available in our scenario. Hierarchical clustering [55] does not need the cluster number, but it requires a distance function for two sub-clusters. It is not easy to derive a correlation distance between two clusters. Thus K-means [54], Gaussian mixture [55], Hierarchical clustering [55] are not applicable in our scenario. DBSCAN [56] only depends on the distance function without the input of cluster number. It has two parameters: *eps* and *minPts*. *eps* specifies the radius of a neighborhood, which controls whether two items would have an edge in the neighbor graph. *minPts*($minPts \geq 2$) specifies the minimum points within distance *eps* of core point. As suggested in [56, 57], we can fix *minPts* by domain knowledge and use the *k-dist* method to determine *eps*. However, *minPts* is hard to determine in practice. So we fix *minPts* to the minimum value 2.

**Additional constraint:** In theory, machines from different modules can be clustered together. Some of these clusters can be intuitively explained, such as data center level network outage, but most of them cannot be intuitively explained. Therefore, we adopted a conservative approach that only machines from the same module would be clustered.

After clustering, the machines are clustered into several groups. Each group is called a *digest*, which consists of a set of machines and a set of KPIs. "M1" and "M2" in Fig. 6 are two example digests.

## IV. Digest Ranking

As shown in Fig. 2, the last phase is to rank digests output by phase 2 (§III). The distilled digests need to be ranked in order to quickly localize the root cause machines. We propose a ranking algorithm to rank the digests so that the one most relevant to the root cause can be listed at the top. Here we adopt Learning-to-rank [58]'s pointwise approach to train a classifier using logistic regression [59]. Learning-to-rank [58] is the application of machine learning in the construction of ranking models. Next we will introduce the features that are used to train the Learning-to-rank model.

We observed that root cause machines have the following characteristics:

**Observation 1:** The change start times of some KPIs of root cause machines are earlier than $T_f$ (see Fig. 1), which means that these KPIs have changes before failure start time. This is reasonable because it takes some times for the failure of the root cause machines to affect the service's critical KPI.

**Observation 2:** The change start times of some KPIs of root cause machines are similar to each other, while the change start times of the KPIs of other machines might not be similar because it takes time for the failure to propagate from root cause machines to other machines and modules and the time taken might be affected by some random factors.

**Observation 3:** Some KPIs of root cause machines have large change degrees.

| Module | Machine | Ratio | KPI | Downward Change (o) | Upward Change (u) |
|---|---|---|---|---|---|
| M1 | DC1_m1_1 ; DC1_m1_2; ...... DC1_m1_27; | 0.036 (27/750) | CPU_HT_IDLE | 45.3 | 0.0 |
| | | | CPU_IDLE | 34.6 | 0.0 |
| | | | CPU_SERVER_LOADAVG_1 | 0.1 | 28.3 |
| | | | CPU_SERVER_LOADAVG_5 | 0.1 | 25.5 |
| | | | CPU_SERVER_LOADAVG_15 | 0.2 | 24.3 |
| | | | NET_TCP_OUT_SEGS | 0 | 22.5 |
| | | | NET_TCP_IN_SEGS | 0 | 21.4 |
| | | | MEM_CACHED | 6.5 | 1.8 |
| | | | ... | ... | ... |
| M2 | DC1_m2_1; ...... DC1_m2_31; DC2_m2_1 ...... DC2_m2_34; | 0.21 (65/312) | MEM_BUFFERS | 21.9 | 4.6 |
| | | | CPU_SERVER_LOADAVG_15 | 0.36 | 9.0 |
| | | | DISK_TOTAL_READ_REQ | 0.45 | 8.6 |
| | | | ... | ... | ... |
| ... | ... | ... | ... | ... | ... |

Fig. 6: The ranked digests output by FluxRank for a failure that 27 machines's CPU got overloaded. DC denotes the data center. Each row represents a digest.

**Observation 4:** The ratio of the number of root cause machines is related to the root cause, because more root cause machines have larger effects.

Next, we will construct the features for each digest according to above observations. Let us introduce the notations firstly. $d$ denotes a digest, and $I$ denotes a machine. We use the superscript to denote the KPI and use the subscript to denote the machine. For example, $o_I^k$ denotes the upward change degree of KPI $k$ on machine $I$.

**Choosing Candidate KPI Set.** Motivated by *Observation 1*, for a digest $d$, we firstly choose KPIs which change before failure start time $T_f$ as the candidate KPI set, denoted by $candidate_set$. More specifically, all features of a digest $d$ are constructed from its $candidate_set$. $T_{c\{I\}}^k$ denotes the change start times of KPI $k$ from the machines $\{I\}$ of digest $d$. We use $\hat{T_c}^k$ to denote the mean of $T_{c\{I\}}^k$. For a KPI $k$, if $\hat{T_c}^k \leq T_f$, then KPI $k$ belongs to the digest's $candidate_set$.

**Feature Extraction.** Firstly, features are extracted from the change start times of KPIs in $candidate_set$. More specifically, motivated by *Observation 1*, we use the maximum, minimum, summation and mean to represent the distribution of $\{\hat{T_c}^k\}$ of all KPIs' in $candidate_set$, denoted by *max_*$T_c$, *min_*$T_c$, *sum_*$T_c$, *mean_*$T_c$.

Secondly, we calculate the standard deviation of $T_{c\{I\}}^k$, denoted by $\hat{std}^k$. If $\hat{std}^k$ is small, it means that KPI $k$ of all the machines in $\{I\}$ have changes together in a short time. Motivated by *Observation 2*, we use the maximum, minimum, summation and mean to represent the distribution of $\{\hat{std}^k\}$ of all KPIs' in $candidate_set$, denoted by *max_std*, *min_std*, *sum_std*, *mean_std*.

Thirdly, for each KPI $k$ in $candidate_set$, we calculate the mean of $o_{\{I\}}^k$, denoted by $\hat{o}^k$, and the mean of $u_{\{I\}}^k$, denoted by $\hat{u}^k$. $\hat{o}^k$ and $\hat{u}^k$ represent KPI $k$'s average change degree of all machines $\{I\}$. Then we use the maximum value of $\{\hat{o}^k, \hat{u}^k\}$ to denote KPI $k$'s maximum change degree, denoted by $max_d^k$. Motivated by *Observation 3*, we use the maximum, minimum, summation and mean to represent the distribution of $\{max_d^k\}$ of all KPIs' in $candidate_set$, denoted by *max_d*, *min_d*, *sum_d*, *mean_d*.

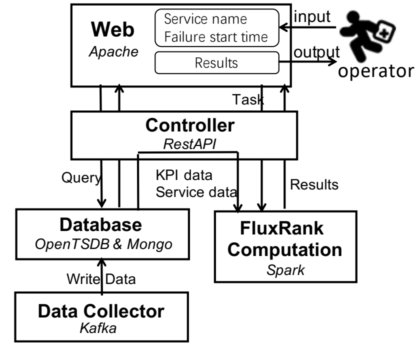


Fig. 7: The implementation of FluxRank

Finally, motivated by *Observation 4*, we calculate the ratio of the machines in the digest $d$ to the total number of machines of the same module, denoted by *ratio*.

We use logistic regression to learn the best linear combination of the features. All training digests are labeled by the operators' mitigation records, and the label details are described in §VI-A.

The KPIs within a digest also need to be ranked so that operators can directly localize the anomalous KPIs. For a digest $d$, all KPIs within $candidate_set$ are ranked by their $max_d^k$ in descending order, and they are ranked ahead of all other KPIs. In other words, the top one KPI of a digest $d$ is the one with the largest change degree in $d$'s $candidate_set$.

Fig. 6 shows the ranked digests for a real failure (see details of the failures in §VI-G) that 27 machines got CPU overload out of 11519 machines. We can see that the top 1 digest contains the 27 root cause machines, and the top 5 KPIs are CPU related KPIs. From FLuxRank's recommended results, Operators can easily understand that 27 machines of module M1 from datacenter DC1 got CPU overload.

## V. System Implementation

Since a large software service has tens of thousands of KPIs, FluxRank is implemented as a highly efficient distributed system, as shown in Fig. 7. It basically has five components: *Web UI*, *Controller*, *FluxRank Computation*, *Database* and *Data Collector*.

*Web UI* displays the data of KPIs and service. When a failure happens, the operator can input the service name and the failure time in the *Web UI* to trigger FluxRank. *Controller* is a key component to connect other components. Its basic function is to read and write the database, trigger the task of FluxRank and store the results to the database, which is implemented with *RestAPI* [60]. *FluxRank Computation* implements FluxRank's change quantification, digest distillation and digest ranking procedures with *Spark* [61]. After a task is submitted, FluxRank can output the recommendation results of faulty machines and related anomalous KPIs within one minute. According to the results, operators could take some actions (*e.g.* switch traffic, reboot) to mitigate the failure. *Database* stores all the KPI data and service meta data (including service, modules, machines, etc.). It is implemented with OpenTSDB [62] and Mongo [63]. *Data Collector* is the agent deployed at each machines to collect all the KPI data. It use *Kafka* [64] to write real-time data to the database.

TABLE II: Failure cases from five real production systems

| Service | #Modules | #Machines | Description | #Case |
|---|---|---|---|---|
| p1 | 29 | 11,519 | an application system for desktop clients that handles billions of user requests per day | 10 |
| p2 | 17 | 2,147 | an application system for mobile clients that handles billions of user requests per day | 48 |
| p3 | 91 | 5,747 | a monitoring system A for the whole company | 7 |
| p4 | 85 | 3,872 | a financial service system like paypal | 1 |
| p5 | 7 | 238 | a monitoring system B for the whole company | 4 |

## VI. Offline Evaluation

In historical section, we present the details of our offline evaluation. 70 real failure cases from five production software services (with tens of thousands of machines) from company $S$ are used to evaluate FluxRank's performance. In §VI-C, we compare different distance functions of clustering algorithm. Then we compare the localization performance of fluxRank with a baseline algorithm in §VI-D. In §VI-E, we analyze the search space reduction of localization. The localization time of FluxRank and manual are compared in §VI-F. Finally, we introduce some typical cases in §VI-G.

### A. Dataset

**Failure Cases:** We collected 70 real cases from five different software services of $S$. The operators recorded the mitigation details of these cases, including when the failure started, what the root causes were, how the failure was mitigated, and the timeline of the whole mitigation processes. Table II summarizes these failure cases.

**KPI Set:** Since KPIs are the fundamental input of the failure localization, KPIs should somehow reflect the root cause of failures. To make our experiment general and easily understood, here we use the standard machine KPIs of the Linux system as the KPI set. Table I shows the 47 standard machine KPIs of the Linux system for the evaluation, consisting of states of CPU, memory, disk, network and OS kernel. The value of the KPIs can be collected from the special files in the */proc* directory of Linux system.

**Labels of Cases:** The operators added two labels to each failure case based on the recorded diagnosis details: *root cause machines* (RCM) and *relevant KPI* (RK).

*Root Cause Machines* (RCM) refers to the machine where the root cause took place. A failure may have multiple RCMs.

*Relevant KPI* (RK) is a KPI relevant to root cause. A failure case usually has multiple RKs. For example, if a machine has a sudden overload of computation tasks, multiple CPU-related KPIs (*e.g.* CPU idle) will reflect sudden changes. Any of these KPIs can help operators to recognize the CPU overload.

### B. Evaluation Methodology

**Root Cause Digest (RCD):** RCD output by FluxRank is a digest satisfying the following conditions: 1) All machines of a digest are RCMs. 2) The top-five KPIs of a digest contain one or more RK(s).

A failure case may have one or multiple RCDs. If the root cause comes from a single module, the case usually has one RCD. Some of the cases have multiple RCDs, in which the clustering algorithm fails to cluster all RCMs together for some single root cause module cases. If the root cause is the failure of a datacenter, *e.g.* power outage, all modules deployed in this datacenter will fail in such a case. Presenting any one of the RCDs to the operators is helpful enough to localize the failure datacenter. Therefore, we consider the result of FluxRank is successful if any of the RCDs is ranked to top. For the 70 failures, FluxRank outputs 83,948 digests in phase 2 and 277 RCDs in phase 3.

TABLE III: Statistical analysis of *internal distance* and *cross distance* using Pearson [51], Kendall's tau [52], Spearman [53], and Euclidean. The *internal distance* represents the distance between RCMs, and the *cross distance* represents the distance between RCMs and non-RCMs.

| | Internal Distance | | Cross Distance | |
|---|---|---|---|---|
| | mean | std | mean | std |
| Pearson | **0.193** | 0.093 | **1.005** | 0.154 |
| Kendall's tau | 0.391 | 0.123 | 1.012 | 0.13 |
| Spearman | 0.357 | 0.125 | 1.012 | 0.14 |
| Euclidean | 34.3 | 19.0 | 86.3 | 25.8 |

### C. Choosing the Distance Function

We use an experiment to compare four distance functions (§III): *Pearson correlation* [51], *Kendall's tau* [52], *Spearman correlation* [53], and *Euclidean distance*.

We randomly sample 10 root cause modules from the 70 cases, and the root cause modules contain both RCMs and non-root cause machines (non-RCMs). A good distance function should produce small *internal distance* between two RCMs, and it should produce large *cross distance* between a RCM and a non-RCM. We do not put any restrictions to the distance between two non-RCMs, because the non-RCMs can be in different states and hence behave differently in their KPIs.

Table III shows the statistical analysis for the *internal distance* and *cross distance*. We find that all four functions produce larger values on cross distances than on internal distances. So all of them are qualified. But we also observe that *Pearson correlation* performs best in terms of relative difference. The average cross distance of *Pearson correlation* is 5 times of the internal distance. For the other three functions, the ratio is around 2 to 3 times. Therefore, we believe *Pearson correlation* is the best one, and apply it in our algorithm.

### D. Baseline Comparison

We choose PAL [65] as our baseline. As discussed in §I, there are many previous works on root cause localization, but most of them rely on the dependency graphs. It is difficult to construct the graphs for the online large distributed software services. Therefore, we only choose PAL [65] as our baseline which does not rely on the dependency graph.

PAL [65] localizes the root cause by sorting the start times of KPIs' anomalous changes. PAL [65] can localize root cause at machine level and output a ranking result, but it does not cluster the machines. So each result item is single machine. During evaluation, for each case, if top k results of PAL [65] contains one or more root cause machines, then we consider it successfully localizes the root cause machine.

TABLE IV: The performances of different models.

| Model | Recall@1 | Recall@2 | Recall@3 |
|---|---|---|---|
| FluxRank(5-fold) | 0.78(55/70) | 0.89(62/70) | 0.94(66/70) |
| FluxRank(3-fold) | 0.78(55/70) | 0.9(63/70) | 0.94(66/70) |
| FluxRank(2-fold) | 0.85(60/70) | 0.89(62/70) | 0.94(66/70) |
| PAL [65] | 0.14(10/70) | 0.21(15/70) | 0.27(19/70) |

As described in §IV, FluxRank applies logistic regression [59] to train a pointwise ranking model. To evaluate its performance, we use *Recall@K*, which is a commonly used metric of ranking. For FluxRank:

$$\textbf{Recall@K} = \frac{\text{\# of cases whose top-k } \textbf{digests} \text{ contain } \textbf{RCDs}}{\text{\# of all cases}}$$

, and the PAL [65] 's *Recall@K* is :

$$\textbf{Recall@K} = \frac{\text{\# of cases whose top-k } \textbf{machines} \text{ contain } \textbf{RCMs}}{\text{\# of all cases}}$$

We apply n-fold cross-validation [66] on our dataset. We applied 2-fold, 3-fold, and 5-fold cross-validations for FluxRank. Table IV shows the results. It is clear that the FluxRank achieves much better performance. Its Recall@3 is 0.94, meaning it ranks 66/70 RCDs cases into top 3. Furthermore, the performance of FluxRank is stable among different folds of cross validations. We can see that the performance of PAL [65] is not good, the reasons are as follows. First, for a large-scale distributed software service, the root cause of a failure is very complex and the root cause machines are related to many features. Simply sorting the start times of KPIs' anomalous changes cannot localize the root cause machines. Second, different KPIs have different collection intervals, the smallest interval can be ten second, and the greatest interval can be one minute or even larger. Therefore, KPIs with different collection intervals cannot be sorted.

### E. Reducing Search Space of Localization

Anomalous machines are the machines which contain one or more anomalous KPIs. Fig. 8 shows the number of anomalous machines of the 70 cases. There are about 80% cases which have more than 300 anomalous machines. Fig. 9 shows that about 80% cases have more than 20% anomalous machines. Compared with screening all the massive anomalous machines, FluxRank can rank the root cause to top-k digests. So operators only need to check top-k digests, which can significantly reduce the operators workload.

### F. Time Cost

The time cost of localization starts from the mitigation start time $T_m$ and ends when the root cause machines are localized. FluxRank is triggered at $T_m$ by operators. When operators obtain FluxRank's ranked digests, they will take few minutes to confirm the results. Therefore, the time cost of localization by FluxRank consist of two parts: running time of FluxRank and manual confirmation time by operators.

We experimented with two servers, each server has 130G memory and 256 cores. The running time of FluxRank is about 1 minute. We set the confirmation time to 5 minutes by operators experience, which is a relative large (thus conservative) value. So the time cost of FluxRank is 6 minutes. because there are only 33 out of 70 cases recorded the manually time cost, we only compare these 33 cases's time cost between manually localization and FluxRank, as shown in Fig. 10. FluxRank reduce the diagnosis time by more than 80% on average.

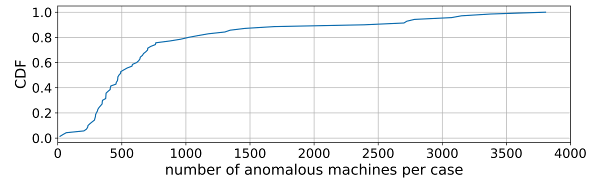


Fig. 8: CDF of the number of anomalous machines of 70 cases

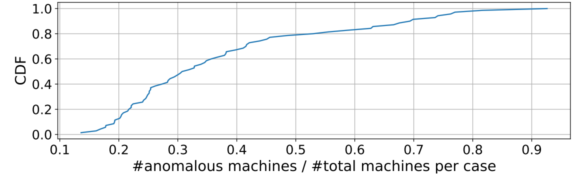


Fig. 9: CDF of anomalous machines ratio of 70 cases

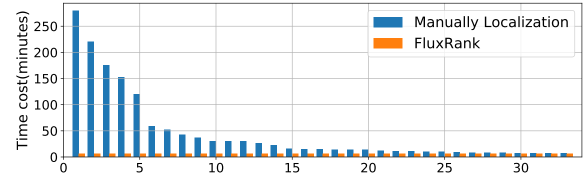


Fig. 10: The time cost of manually localiztion and FluxRank for 33 failure cases.

### G. Case Study

Due to the limited space, we only introduce in Table II three representative cases from the 70 failure causes, whose RCDs are ranked as top one, top two and out of top three.

**RCD ranked the first.** This case is a *CPU overload* failure. The service runs on 11,519 machines (p1 in Table II). The root cause of this failure is a faulty configuration that causes CPU overload on 27 machines. In this case, the testers were doing some offline stress tests. Due to an operator's faulty configuration, the CPUs of some online machines were overloaded. After a few minutes, on-call operators found that the response time of the service is anomalous. Then, the on-call operators tried to mitigate the failure according to their experience. They spent about one hour, with no success. Then, this failure was escalated, and the operators stopped all stress tests that may influence online service. Eventually, they successfully mitigated the failure, but spent about two hours in total. When the operators discussed this failure later on, they found that the Root Cause Machines are 27 machines whose CPUs got overloaded. Fig. 6 shows the output of FluxRank for this case, FluxRank successfully recommends the RCD at top one. The recommended RCD contains the 27 anomalous machines, and the top 5 anomalous KPIs are all CPU-related KPIs, belonging to the first row in Table I.

**RCD ranked second.** This case is an *insufficient resource* failure. The service, which is the major monitoring service of $S$, runs on 5,747 machines (p3 in Table II). The root cause is that module A has insufficient resource to handle the increased traffic. Module B has strong dependencies on module A. So when module A's machines become anomalous, module B's machines also become anomalous. The features of module B's digest is very similar with module A's digest. As FluxRank does not explicitly analyze the dependencies between modules, the digest of module B is ranked first, and that of module

A (the RCD) is ranked second. This case is representative of all cases whose RCDs are ranked as top 2 and top 3 by FluxRank. We plan to investigate incorporating dependencies into FluxRank in a widely-deployable way in the future.

**RCD is not ranked into top three.** This case is a *memory overload* failure. The service, which is the major mobile business service that handles billions of user requests per day, runs on 2147 machines (p2 in Table II). The reason why the RCD is not ranked into top three is that this service is deployed on many VMs, and the resource KPIs of the VMs are not monitored. Therefore, the sudden changes on resource KPIs of these VMs cannot be clearly represented by the physical machines' resource KPIs they are on. The lesson we learned is that the VM KPI should also be monitored, which can enable FluxRank to successfully localize such failure in the future. On the other hand, analyzing similar cases helps identify KPIs that should have been monitored in a very targeted manner and gradually helps improve the coverage the monitoring.

## VII. Online Deployment

FluxRank has been successfully deployed online on one Internet service (with hundreds of machines) in $S$ and six banking services (each with tens of machines) in two large banks for three months. Table V shows the details of the 7 real services and the online performance of FluxRank. A *valid case* is one whose RCD is ranked first. We can see that FluxRank ranked the RCDs to top-1 for 55 cases in the 59 online cases. Because the RCDs of other 4 cases are not ranked into top 3, so Table V only shows the top-1 results. Next we introduce the details of some representative valid cases.

**Valid Case 1:** The root cause of this case (in the Internet service) is that the number of database reading operations of a database module increased. When operators found the response time KPI of the service is anomalous, they used FluxRank to localize the root cause machines. The top one digest of the recommendation result contains 4 machines and 16 anomalous KPIs. Operators observed that these 4 machines came from the database module, and the top five anomalous KPIs are all related to the database reading operations. Therefore, they quickly and successfully mitigated the failure by switching from these 4 machines to backup database servers.

**Valid Case 2:** On Oct. 26, 2018, one banking service's number of failed transaction increased significantly, which is a failure. The root cause is that someone started a process occupying most of CPU and memory of the web-proxy machines. FluxRank quickly localized the root cause machines within 3 mins. The top one digest of the recommended RCD showed the web-proxy machines's CPU and memory usage became anomalous. So the operator stopped the process and the banking service returned to normal.

There are 4 cases that FluxRank did not recommend the RCDs at the top. The reasons are: 1) The root cause of the failure, *e.g.*, the business logic error, is not reflected in the KPIs, This is the limitation of FluxRank, which only utilizes monitored machine KPIs. We plan to improve on this limitation in the future. 2) For the problems in data collection system, there are many missing data in the KPIs. The lessons are that the underlying monitoring system needs to be robust in order for FluxRank to work reliably.

TABLE V: The details of 59 online cases from 7 real services. The valid case represents the case whose RCD is ranked first.

| | **#module** | **#machine** | **#KPI/machine** | **(#valid case) / (#total case)** |
|---|---|---|---|---|
| s1 | 15 | 520 | 591 | 2/2 |
| s2 | 3 | 30 | 120 | 1/1 |
| s3 | 4 | 40 | 302 | 3/3 |
| s4 | 4 | 38 | 520 | 3/5 |
| s5 | 3 | 35 | 424 | 1/1 |
| s6 | 4 | 38 | 512 | 3/5 |
| s7 | 7 | 26 | 311 | 42/42 |

## VIII. Related Work

For a large-scale distributed software service, diagnosing failure is a challenging and classic research topic. §I has overviewed the dependence-based failure diagnosis work, and below we provide other representative related works.

**Log-based troubleshooting:** System log is an important clue for diagnosis [67–70]. Works [71, 72] train the normal pattern from history data and use these patterns to detect the anomalies. For large-scale service, the volume of logs is huge. Directly analyzing the log has a high overhead. FluxRank is light-weight because it only focuses on the KPI data.

**Trace-based troubleshooting:** Pinpoint [73] and Depper [74] collect the execution path information (traces). When failure happens, traces are useful to localize the root cause. But works [73, 74] need to modify the source code of service. Generally, a large-scale service is developed by many teams with different languages over the years, the overhead of modifying source code is often too high. FluxRank can be easily deployed without modifying any source code.

## IX. Conclusion

This paper presents FluxRank, a widely-deployable framework that can automatically and accurately localize the root cause machines underlying software service failures, so that some actions can be triggered to mitigate the service failure. Our evaluation using historical cases from five real services (with tens of thousands of machines) show that the true faulty machines are ranked top 1 (top 3) for 55 (66) cases out of 70 cases. Compared to existing approach FluxRank cuts the localization time by more than 80% on average (to less than 6 minutes). FluxRank has been successfully deployed online on one Internet service (with hundreds of machines) and six banking services (each with tens of machines) for three months, and correctly localized the root cause machines as the top 1 for 55 cases out of 59 cases.

In the future, we plan to extend the FluxRank's framework to include the text-based machine logs, also widely available but more challenging to analyze, to provide more detailed localization information within the identified faulty machines.


## Acknowledgment

The authors gratefully acknowledge the contribution of Juexing Liao and Fuying Wang for proofreading this paper. This work has been supported by the Beijing National Research Center for Information Science and Technology (BNRist) key projects, the Fundamental Research Funds for the Central Universities (Grant No. 63191427), and CERNET Innovation Project (Grant No. NGII20180121).

# References


[1] L. Hook, "Amazon failure disrupts hundreds of thousands of websites." https://www.ft.com/content/b809c752-fded-11e6-96f8-3700c5664d30, march 1, 2017.

[2] J. NOVET, "Microsoft confirms azure storage issues around the world." https://venturebeat.com/2017/03/15/microsoft-confirms-azure-storage-issues-around-the-world, march 15, 2017.

[3] S. Zhang, Y. Liu, D. Pei, Y. Chen, X. Qu, S. Tao, and Z. Zang, "Rapid and robust impact assessment of software changes in large internet-based services," in *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*. ACM, 2015, p. 2.

[4] S. Zhang, Y. Liu, D. Pei, Y. Chen, X. Qu, S. Tao, Z. Zang, X. Jing, and M. Feng, "Funnel: Assessing software changes in web-based services," *IEEE Transactions on Service Computing*, 2016.

[5] Y. Sun, Y. Zhao, Y. Su, D. Liu, X. Nie, Y. Meng, S. Cheng, D. Pei, S. Zhang, X. Qu *et al.*, "Hotspot: Anomaly localization for additive kpis with multi-dimensional attributes," *IEEE Access*, vol. 6, pp. 10 909–10 923, 2018.

[6] D. Bernstein, "Containers and cloud: From lxc to docker to kubernetes," *IEEE Cloud Computing*, no. 3, pp. 81–84, 2014.

[7] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center." in *NSDI*, vol. 11, no. 2011, 2011, pp. 22–22.

[8] P. Bahl, R. Chandra, A. Greenberg, S. Kandula, D. A. Maltz, and M. Zhang, "Towards highly reliable enterprise network services via inference of multi-level dependencies," in *ACM SIGCOMM Computer Communication Review*, vol. 37, no. 4. ACM, 2007, pp. 13–24.

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

[10] P. Chen, Y. Qi, P. Zheng, and D. Hou, "Causeinfer: automatic and distributed performance diagnosis with hierarchical causality graph in large distributed systems," in *INFOCOM, 2014 Proceedings IEEE*. IEEE, 2014, pp. 1887–1895.

[11] H. Nguyen, Z. Shen, Y. Tan, and X. Gu, "Fchain: Toward black-box online fault localization for cloud systems," in *Distributed Computing Systems (ICDCS), 2013 IEEE 33rd International Conference on*. IEEE, 2013, pp. 21–30.

[12] X. Nie, Y. Zhao *et al.*, "Mining causality graph for automatic web-based service diagnosis," in *Performance Computing and Communications Conference (IPCCC), 2016 IEEE 35th International*. IEEE, 2016, pp. 1–8.

[13] S. Kandula, R. Mahajan, P. Verkaik, S. Agarwal, J. Padhye, and P. Bahl, "Detailed diagnosis in enterprise networks," *ACM SIGCOMM Computer Communication Review*, vol. 39, no. 4, pp. 243–254, 2009.

[14] T. Ahmed, B. Oreshkin, and M. Coates, "Machine learning approaches to network anomaly detection," in *Proceedings of the 2nd USENIX Workshop on Tackling Computer Systems Problems with Machine Learning Techniques*. USENIX Association, 2007, pp. 1–6.

[15] Y. Liu, L. Zhang, and Y. Guan, "A distributed data streaming algorithm for network-wide traffic anomaly detection," *ACM SIGMETRICS Performance Evaluation Review*, vol. 37, no. 2, pp. 81–82, 2009.

[16] R. Jiang, H. Fei, and J. Huan, "Anomaly localization for network data streams with graph joint sparse pca," in *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2011, pp. 886–894.

[17] J. Gao, G. Jiang, H. Chen, and J. Han, "Modeling probabilistic measurement correlations for problem determination in large-scale distributed systems," in *2009 29th IEEE International Conference on Distributed Computing Systems*. IEEE, 2009, pp. 623–630.

[18] C. Wang, I. A. Rayan, G. Eisenhauer, K. Schwan, V. Talwar, M. Wolf, and C. Huneycutt, "Vscope: middleware for troubleshooting time-sensitive data center applications," in *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer, 2012, pp. 121–141.

[19] M. Kalisch and P. Bühlmann, "Estimating high-dimensional directed acyclic graphs with the pc-algorithm," *Journal of Machine Learning Research*, vol. 8, no. Mar, pp. 613–636, 2007.

[20] S.-B. Lee, D. Pei *et al.*, "Threshold compression for 3g scalable monitoring," in *INFOCOM, 2012 Proceedings IEEE*. IEEE, 2012, pp. 1350–1358.

[21] A. H. Yaacob, I. K. Tan *et al.*, "Arima based network anomaly detection," in *Communication Software and Networks, 2010. ICCSN'10. Second International Conference on*. IEEE, 2010, pp. 205–209.

[22] H. Yan, A. Flavel *et al.*, "Argus: End-to-end service anomaly detection and localization from an isp's point of view," in *INFOCOM, 2012 Proceedings IEEE*. IEEE, 2012, pp. 2756–2760.

[23] B. Krishnamurthy, S. Sen, Y. Zhang, and Y. Chen, "Sketch-based change detection: methods, evaluation, and applications," in *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*. ACM, 2003, pp. 234–247.

[24] F. Knorn and D. J. Leith, "Adaptive kalman filtering for anomaly detection in software appliances," in *INFOCOM Workshops 2008, IEEE*. IEEE, 2008, pp. 1–6.

[25] A. Mahimkar, Z. Ge *et al.*, "Rapid detection of maintenance induced changes in service performance," in *Proceedings of the Seventh COnference on emerging Networking EXperiments and Technologies*. ACM, 2011, p. 13.

[26] D. Liu, Y. Zhao *et al.*, "Opprentice: Towards practical and automatic anomaly detection through machine learning," in *Proceedings of the 2015 ACM Conference on Internet Measurement Conference*, ser. IMC '15. New York, NY, USA: ACM, 2015, pp. 211–224.

[27] H. Xu, W. Chen *et al.*, "Unsupervised anomaly detection via variational auto-encoder for seasonal kpis in web applications," in *Proceedings of the 2018 World Wide Web Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 2018, pp. 187–196.

[28] J. Bu, Y. Liu, S. Zhang, W. Meng, Q. Liu, X. Zhu, and D. Pei, "Rapid deployment of anomaly detection models for large number of emerging kpi streams," in *2018 IEEE 37th International Performance Computing and Communications Conference (IPCCC)*. IEEE, 2018, pp. 1–8.

[29] M. Ma, S. Zhang, D. Pei, X. Huang, and H. Dai, "Robust and rapid adaption for concept drift in software system anomaly detection," in *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2018, pp. 13–24.

[30] S. Reddy, M. Mun, J. Burke, D. Estrin, M. Hansen, and M. Srivastava, "Using mobile phones to determine transportation modes," *ACM Transactions on Sensor Networks (TOSN)*, vol. 6, no. 2, p. 13, 2010.

[31] Y. Zheng, Y. Chen, Q. Li, X. Xie, and W.-Y. Ma, "Understanding transportation modes based on gps data for web applications," *ACM Transactions on the Web (TWEB)*, vol. 4, no. 1, p. 1, 2010.

[32] I. Cleland, M. Han, C. Nugent, H. Lee, S. McClean, S. Zhang, and S. Lee, "Evaluation of prompted annotation of activity data recorded from a smart phone," *Sensors*, vol. 14, no. 9, pp. 15 861–15 879, 2014.

[33] M. Han, Y.-K. Lee, S. Lee *et al.*, "Comprehensive context recognizer based on multimodal sensors in a smartphone," *Sensors*, vol. 12, no. 9, pp. 12 588–12 605, 2012.

[34] K. D. Feuz, D. J. Cook, C. Rosasco, K. Robertson, and M. Schmitter-Edgecombe, "Automated detection of activity transitions for prompting," *IEEE transactions on human-machine systems*, vol. 45, no. 5, pp. 575–585, 2015.

[35] F. Gustafsson and F. Gustafsson, *Adaptive filtering and change detection*. Citeseer, 2000, vol. 1.

[36] F. Desobry, M. Davy, and C. Doncarli, "An online kernel change detection algorithm," *IEEE Transactions on Signal Processing*, vol. 53, no. 8, pp. 2961–2974, 2005.

[37] E. Keogh and J. Lin, "Clustering of time-series subsequences is meaningless: implications for previous and future research," *Knowledge and information systems*, vol. 8, no. 2, pp. 154–177, 2005.

[38] A. Aue, S. Hörmann, L. Horváth, M. Reimherr *et al.*, "Break detection in the covariance structure of multivariate time series models," *The Annals of Statistics*, vol. 37, no. 6B, pp. 4046–4087, 2009.

[39] K. Yamanishi and J.-i. Takeuchi, "A unifying framework for detecting outliers and change points from non-stationary time series data," in *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2002, pp. 676–681.

[40] M. Yamada, A. Kimura, F. Naya, and H. Sawada, "Change-point detection with feature selection in high-dimensional time-series data." in *IJCAI*, 2013, pp. 1827–1833.

[41] D. Barry and J. A. Hartigan, "A bayesian analysis for change point problems," *Journal of the American Statistical Association*, vol. 88, no. 421, pp. 309–319, 1993.

[42] X. Xuan and K. Murphy, "Modeling changing dependency structure in multivariate time series," in *Proceedings of the 24th international conference on Machine learning*. ACM, 2007, pp. 1055–1062.

[43] P. R. Rosenbaum, "An exact distribution-free test comparing two multivariate distributions based on adjacency," *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, vol. 67, no. 4, pp. 515–530, 2005.

[44] J. Zhang and M. Small, "Complex network from pseudoperiodic time series: Topology versus dynamics," *Physical review letters*, vol. 96, no. 23, p. 238701, 2006.

[45] L. Lacasa, B. Luque, F. Ballesteros, J. Luque, and J. C. Nuno, “From time series to complex networks: The visibility graph,” *Proceedings of the National Academy of Sciences*, vol. 105, no. 13, pp. 4972–4975, 2008.
[46] D. Rybach, C. Gollan, R. Schluter, and H. Ney, “Audio segmentation for speech recognition using segment features,” in *Acoustics, Speech and Signal Processing, 2009. ICASSP 2009. IEEE International Conference on*. IEEE, 2009, pp. 4197–4200.
[47] P. Fearnhead, “Exact and efficient bayesian inference for multiple changepoint problems,” *Statistics and computing*, vol. 16, no. 2, pp. 203–213, 2006.
[48] P. Wilmott, S. Howson, S. Howison, J. Dewynne *et al.*, *The mathematics of financial derivatives: a student introduction*. Cambridge university press, 1995.
[49] B. W. Silverman, *Density estimation for statistics and data analysis*. CRC press, 1986, vol. 26.
[50] D. C. Montgomery, G. C. Runger, and N. F. Hubele, *Engineering statistics*. John Wiley & Sons, 2009.
[51] J. Benesty, J. Chen *et al.*, “Pearson correlation coefficient,” in *Noise reduction in speech processing*. Springer, 2009, pp. 1–4.
[52] M. G. Kendall, “A new measure of rank correlation,” *Biometrika*, vol. 30, no. 1/2, pp. 81–93, 1938.
[53] W. Pirie, “Spearman rank correlation coefficient,” *Encyclopedia of statistical sciences*, 1988.
[54] J. A. Hartigan and M. A. Wong, “Algorithm as 136: A k-means clustering algorithm,” *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, vol. 28, no. 1, pp. 100–108, 1979.
[55] L. Rokach and O. Maimon, “Clustering methods,” in *Data mining and knowledge discovery handbook*. Springer, 2005, pp. 321–352.
[56] M. Ester, H.-P. Kriegel, J. Sander, X. Xu *et al.*, “A density-based algorithm for discovering clusters in large spatial databases with noise.” in *Kdd*, vol. 96, no. 34, 1996, pp. 226–231.
[57] D. Y. Yoon, N. Niu, and B. Mozafari, “Dbsherlock: A performance diagnostic tool for transactional databases,” in *Proceedings of the 2016 International Conference on Management of Data*. ACM, 2016, pp. 1599–1614.
[58] T.-Y. Liu *et al.*, “Learning to rank for information retrieval,” *Foundations and Trends® in Information Retrieval*, vol. 3, no. 3, pp. 225–331, 2009.
[59] F. E. Harrell, “Ordinal logistic regression,” in *Regression modeling strategies*. Springer, 2001, pp. 331–343.
[60] *RestAPI*, accessed Jan. 11, 2019, https://www.restapitutorial.com/.
[61] *Spark*, accessed Jan. 11, 2019, https://spark.apache.org/.
[62] *OpenTSDB*, accessed April 22, 2018, http://opentsdb.net/.
[63] *MongoDB*, accessed Jan. 11, 2019, https://www.mongodb.com/.
[64] *Kafka*, accessed Jan. 11, 2019, https://kafka.apache.org/.
[65] H. Nguyen, Y. Tan, and X. Gu, “Pal: P ropagation-aware a nomaly l ocalization for cloud hosted distributed applications,” in *Managing Large-scale Systems via the Analysis of System Logs and the Application of Machine Learning Techniques*. ACM, 2011, p. 1.
[66] R. Kohavi *et al.*, “A study of cross-validation and bootstrap for accuracy estimation and model selection,” in *Ijcai*, vol. 14, no. 2. Montreal, Canada, 1995, pp. 1137–1145.
[67] W. Meng, Y. Liu, S. Zhang, D. Pei, H. Dong, L. Song, and X. Luo, “Device-agnostic log anomaly classification with partial labels,” in *2018 IEEE/ACM 26th International Symposium on Quality of Service (IWQoS)*. IEEE, 2018, pp. 1–6.
[68] S. Zhang, Y. Liu, W. Meng, Z. Luo, J. Bu, S. Yang, P. Liang, D. Pei, J. Xu, Y. Zhang *et al.*, “Prefix: Switch failure prediction in datacenter networks,” *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 2, no. 1, p. 2, 2018.
[69] S. Zhang, W. Meng, J. Bu, S. Yang, Y. Liu, D. Pei, J. Xu, Y. Chen, H. Dong, X. Qu *et al.*, “Syslog processing for switch failure diagnosis and prediction in datacenter networks,” in *2017 IEEE/ACM 25th International Symposium on Quality of Service (IWQoS)*. IEEE, 2017, pp. 1–10.
[70] W. Meng, Y. Liu, Y. Zhu, S. Zhang, D. Pei, Y. Liu, Y. Chen, R. Zhang, S. Tao, P. Sun, and R. Zhou, “Loganomaly: Unsupervised detection of sequential and quantitative anomalies in unstructured logs,” in *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*. International Joint Conferences on Artificial Intelligence Organization, 7 2019, pp. 4739–4745.
[71] Q. Fu, J.-G. Lou *et al.*, “Execution anomaly detection in distributed systems through unstructured log analysis,” in *Data Mining, 2009. ICDM'09. Ninth IEEE International Conference on*. IEEE, 2009, pp. 149–158.
[72] W. Xu, L. Huang *et al.*, “Detecting large-scale system problems by mining console logs,” in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 2009, pp. 117–132.
[73] M. Y. Chen, E. Kiciman *et al.*, “Pinpoint: Problem determination in large, dynamic internet services,” in *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*. IEEE, 2002, pp. 595–604.
[74] B. H. Sigelman, L. A. Barroso *et al.*, “Dapper, a large-scale distributed systems tracing infrastructure,” Technical report, Google, Inc, Tech. Rep., 2010.