Rapid and Robust Impact Assessment of Software Changes in Large Internet-based Services

Shenglin Zhang[†§*], Ying Liu[†*], Dan Pei[‡*]
Yu Chen[§], Xianping Qu[§], Shimin Tao[§], Zhi Zang[§]

[†]Institute for Network Sciences and Cyberspace, Tsinghua University
[‡] Department of Computer Science and Technoogy, Tsinghua University
[*] Tsinghua National Laboratory for Information Science and Technology (TNList)
[§]Baidu, Inc.

zhangsl12@mails.tsinghua.edu.cn, liuying@cernet.edu.cn, peidan@tsinghua.edu.cn
chengyu034@hotmail.com, {quxianping, taoshimin, zangzhi}@baidu.com

ABSTRACT

The detection of performance changes in software change roll-outs in Internet-based services is crucial for an operations team, because it allows timely roll-back of a software change when performance degrades unexpectedly. However, it is infeasible to manually investigate millions of performance measurements of many roll-outs.

In this paper, we present an automated tool, FUNNEL, for rapid and robust impact assessment of software changes in large Internet-based services. FUNNEL automatically collects the related performance measurements for each software change. To detect significant performance behavior changes, FUNNEL adopts singular spectrum transform (SST) algorithm as the core algorithm, uses various techniques to improve its robustness and reduce its computational cost, and applies a difference-in-difference (DiD) method to differentiate the true causality from the random correlations between the performance change and the software change. Evaluation through historical data in real-word services shows that FUNNEL achieves an accuracy of more than 99.8%. Compared with previous methods, FUNNEL's detection delay is 38.02% to 64.99% shorter, and its computation speed is 4.59 - 7098 times faster. In real deployment, FUNNEL achieves a 98.21% precision, high robustness, fast detection speed, and shows its capability in detecting unexpected performance changes.

CoNEXT '15, December 01-04, 2015, Heidelberg, Germany
© 2015 ACM. ISBN 978-1-4503-3412-9/15/12$15.00.
DOI: http://dx.doi.org/10.1145/2716281.2836087

Keywords

Software Change, Performance Change, Singular Spectrum Transform, Difference in Difference

1. INTRODUCTION

In large Internet-based services such as search engines, online shopping, and social networking, the operations team needs to frequently conduct software changes, *i.e.*, software upgrades and configuration changes, in order to deploy new features, fix bugs, and improve service performance. Although each software change is extensively tested on testbeds before deployment, errors and bugs may still occur in the operational environment because of diverse hardware/software systems, complex interactions, and the large scale of devices [1, 4]. Therefore, the operations team typically deploys software changes using a "Dark Launching" [2] approach. Instead of rolling out a software change to all servers at one time, the operations team deploys the software change on a subset of servers at the beginning and continuously monitors a predefined list of **K**ey **P**erformance **I**ndicators (KPIs) to determine the impact of the software change. The KPIs cover a wide range of performances, including user-perceived issues (*e.g.*, Web page response delay), service performance (*e.g.*, advertisement click count), hardware health (*e.g.*, server memory utilization), and so on.

If the KPIs on the server subset perform as expected, the software change will be rolled out to all servers. Otherwise, the software change should be rolled back as soon as possible. Generally, service performance degradation may induce poor user experience [1, 4] or revenue drop [8].

Thus it is important to detect significant KPI changes rapidly, whether positive or negative, to allow a timely roll-back. However, the operations team generally assesses the impact of software changes manually in Internet-based services, which has been demonstrated to be error-prone, cumbersome, and almost impossible to scale to a larger size. As a result, some critical issues caused by software changes may fly under the operations team's radar, hurting the applications performance and user experience [1, 4].

In this study, we focus on software changes and their impact on KPIs. Our objective is to build an *automated* tool that detects behavior changes *rapidly* and *accurately* in a broad range of KPIs after a software change, and *accurately* determines whether the behavior changes are caused by the software change. Usually, KPI changes caused by software changes include level shifts, *e.g.*, a sudden increase in memory utilization, or ramp-ups/ramp-downs, *e.g.*, a deteriorating condition. Similar to [20], we focus on level shifts and ramp up/downs induced by software changes in this paper. How to reduce the detection delay, *i.e.*, the delay between the occurrence of a KPI change and its detection is a real challenge for Internet-based services for both *scalability* and *robustness* reasons. First, the impact of a software change can be observed in any of the huge number of KPIs of the services and servers that share the same spatial scope with the software change. In our studied scenarios, there are hundreds, even thousands, of KPIs that should be monitored after each single software change that has occurred, while there are tens of thousands of software changes occurring every day in our scenario. This forces the operations team to monitor several million KPIs every day. The large scale of the problem calls for an algorithm with low computation overhead. Second, with the requirement of short detection delay, we do not have the luxury of using data smoothing and aggregation to achieve robustness any more, and the behavior change detection method should be quite robust [18].

To the best of our knowledge, in the literature there exists no rapid and robust method for impact assessment of software changes *in large Internet-based services*. There exists several studies of impact assessment in the area of network infrastructure [18, 20]. However, the CUmulative SUM (CUSUM) used in [20] suffers from long detection delay [18], because the cumulative sum may take a long time before it exceeds the threshold. Multiscale Robust Local Subspace (MRLS) method applied in [18] achieves low detection delay, but the iteration of Singular Value Decomposition (SVD) used in subspace computation with l_1-norm exhibits high computational complexity [17]. While it works in [18] for thousands of time-series metrics in backbone networks, MRLS would spend too much computational resources for millions of KPIs in Internet-based services, and is hence not feasible in our scenario. The iteration of SVD is essential to MRLS for improving robustness, and it is hardly possible to reduce the computation overhead of MRLS.

Singular Spectrum Transform (SST) [13] has emerged as a popular performance change detection method recently. SST has been demonstrated to be accurate with short detection delay [12, 22]. However, based on SVD, SST still suffers from high computational cost [14], and its accuracy degrades fast in the face of noises [21]. KPIs in Internet-based services are quite diverse intrinsically, exhibiting various characteristics including strong seasonality (*e.g.*, Web page view count), high variability (*e.g.*, server CPU context switch count), and stationarity (*e.g.*, server memory utilization). In addition, the baseline used to compare the performance after software changes may be contaminated by the impact of previous software changes and/or other factors, as is the case in large infrastructure networks [18].

In this paper, we designed and implemented FUNNEL, an automated tool for assessing the impact of software changes rapidly and robustly in large Internet-based services. For a given software change, FUNNEL automatically determines the correct spatial scope of the impact, collects a broad range of KPIs, detects KPI changes, and determines the KPI changes caused by software changes. FUNNEL adopts matrix compression and implicit inner product calculation to reduce SST's computation overhead. The short detection delay and the low computational cost of the improved SST make timely mitigation possible when FUNNEL is deployed online. Furthermore, FUNNEL improves the robustness for SST, making it work quite well across diverse types of KPIs, especially variable KPIs.

In addition to software changes, other factors including seasonality, network hardware breakdowns, malicious attacks, *etc.*, can also give rise to KPI changes. The impact of other factors can over-shadow the assessment of software changes [19]. To achieve the accurate inference of the impact of software changes, it is non-trivial for the operations team to exclude KPI changes caused by other factors. However, neither CUSUM and MRLS, nor the improved SST can exclude the KPI changes induced by other factors. FUNNEL uses a classic method, difference in difference (DiD) [6, 26, 27], to solve the problem by comparing the relative performance between the treated group and the control group. DiD compares the KPIs of service processes and servers in which the software changes have been conducted (hereafter, collectively referred to as *tservers/tinstances*, where "t" stands for "treated") with the KPIs of service processes and servers of the same service (a server is usually dedicated to a specific service in our context) without the software change (hereafter, collectively referred to as *cservers/cinstances*, where "c" stands for "control"). If the operations team conducts the software change without using the Dark Launching method, and there is no server/process in *cservers/cinstances*, FUNNEL compares the impacted KPIs related to the software change with historical measurements of KPIs to exclude seasonality. FUNNEL adopts a relatively long baseline to address baseline contamination (in our prototype implementation, KPIs of 30 days before the day of software change are used to construct the baseline) [18]. Moreover, the large number of KPIs of *cservers/cinstances* also alleviates the impact of baseline contamination.

Our main contributions can be summarized as follows:

(1) We identify the problem of rapid, robust, and automated impact assessment of software changes *in large Internet-based services*, and its research challenges in terms of scal-

ability, robustness, detection delay, and computation cost.

(2) We propose FUNNEL, the first approach in the literature that addresses the above challenges. The core idea of FUNNEL is to use SST, a rapid change point detection algorithm, as our algorithm basis. To reduce SST's computation overhead, FUNNEL adopts matrix compression and implicit inner product calculation. To improve the robustness for SST against noises, FUNNEL uses a DiD method to compare with various control groups.

(3) Evaluation through historical data shows that FUNNEL performs significantly better than CUSUM [20] and MRLS [18], and the operational experiences of real deployment show FUNNEL's good performance and value. We conducted extensive evaluations using manually labeled KPIs collected from 144 different software changes in real-world Internet-based services. Specifically, FUNNEL achieved an accuracy of more than 99.8%, a detection delay that is 58.82% of that of MRLS and 33.76% of that of CUSUM. In addition, FUNNEL is more than 7000 times faster than MRLS in computational speed, and over 4 times faster than CUSUM. Furthermore, we have deployed the prototype of FUNNEL to dozens of real-world Internet-based services. FUNNEL achieved a 98.21% precision based on one week's observation. A few representative cases were presented to show FUNNEL's robustness, detection speed, and capability to detect unexpected KPI changes. In one specific case, FUNNEL reduced the detection delay of one important incident to less than 10 minutes, far less than the1.5 hour used in manual assessment.

The rest of the paper is organized as follows. We provide an introduction to software change and KPI in §2, and describe the design of FUNNEL in §3. The evaluation of FUNNEL is presented in §4, followed by a description of the deployment of FUNNEL and case studies in §5. Finally, we review related works in §6 and conclude our paper in §7.

2. SOFTWARE CHANGES AND KPIS

In this section, we provide a brief introduction of software changes and KPIs.

2.1 Scope of Studied Software Changes

In this paper, we focus on two types of software changes on servers in large Internet-based services, *software upgrades* and *configuration changes*, for the following three reasons. (1) The operations team typically cares about the unexpected consequences that are potentially due to these planned changes; (2) These changes are *controllable* by the operations team via command line interfaces and *observable* in logs; (3) We have observed that these two types constitute the vast majority of the tens of thousands of software changes in our data.

Software Upgrades. With the current rapid evolution of the Internet, new features are continuously being deployed with software upgrades. The operations team also conducts software upgrades to fix bugs or improve service performance. In a large service, it is often the case that one software upgrade implements multiple features or bug fixes, and FUNNEL considers such a software upgrade as a whole. FUNNEL decides whether the whole software upgrade introduces any KPI change but does not attempt to distinguish which individual feature or bug fix introduces KPI changes.

Configuration Changes. Using command line interfaces, the operations team can change the configurations by using specific commands. The configuration change can be in the operating system (OS) or infrastructure software (*e.g.*, a configuration change in Apache), service configuration (*e.g.*, an increase in the number of threads in a service process), deployment scale (*e.g.*, an increase in the number of servers where a service is deployed), or data source (*e.g.*, an update to the strategy that calculates the valid page view counts).

With the above focuses, the following perspectives are out of scope for this paper. (1) We do not consider the software changes on the network devices such as routers and switches, which have been already studied in depth in [18, 19, 20]; (2) We do not consider software changes that were external to the company, *e.g.*, a change in a peer company, since these changes might be invisible to the studied company's operations team. (3)We do not explicitly consider the interactions across multiple concurrent or consecutive software changes on a same server, which can be considered as one combined change as a straw man approach. More detailed studies along the last three directions are left as future work.

2.2 KPI

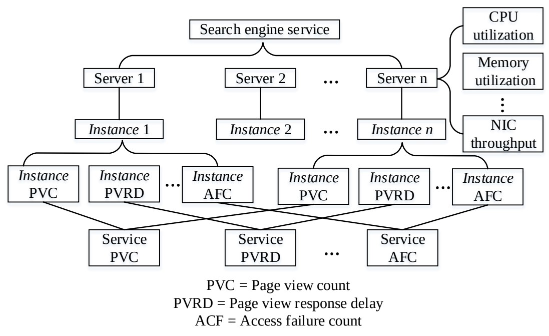

Figure 1: The relationship among service, server, *instance* and KPI

In the studied Internet-based services, there are hundreds of thousands of servers providing various types of services. Each service (*e.g.*, search, web mail, social networking) runs on one or more servers with a specific process on each server. An *instance* denotes a *process* of a specific service on a specific server. A KPI is a performance metric of *a given server/service/process*. There are three types of KPIs that need to be monitored for software changes assessment: server KPIs, *instance* KPIs and service KPIs. The operations team deploys an agent on each server to monitor the status of each *instance* and collect the KPIs of all *instances* continuously. For example, immediately after the process serves a customer with some Web page view, the page view count is incremented and a new page view response delay is recorded. In addition, by analyzing server log files that record the system status, the agent is able to periodically collect server KPIs, such as CPU utilization, memory utilization, and NIC throughput. A service KPI is an aggregation of all *instance* KPIs in the service. Fig. 1 shows an example of the relationship among service, server, *instance* and KPI.

After collecting the measurements of KPIs of servers and *instances*, the agent on each server delivers the measurements to a centralized Hadoop-based database, which also stores the service KPIs aggregated based on the KPIs of the *instances*. The database also provides a subscription tool for other systems, such as FUNNEL, to periodically receive the subscribed measurements based on the server, *instance*, and service. The data collection interval at the servers is typically 1 minute. Within one second, the measurements subscribed by FUNNEL are pushed to FUNNEL.

In large services, there might exist some KPIs of dubious quality. To the best of our knowledge, there is no previous work on eliminating low-quality KPIs in Internet-based services. In this paper, we do not focus on eliminating low-quality KPIs either. FUNNEL detects all KPI changes in the impact set regardless of the quality of the KPI, and delivers the results to the operations team. The operations team will then determine whether the performance changes in the low-quality KPIs are induced by the software change or not.

2.3 KPI changes

A KPI change is defined as a non-transient change (*e.g.*, lasting more than 7 minutes) in a KPI that is introduced by a software change. In this paper, we focus on behavior changes evidenced by individual KPI time series. As Fig. 2 shows, the changes can be either level-shifts immediately after software changes, or ramp ups or downs that ensue gradually over time after the software changes.

Behavior changes in KPIs can validate the expected impacts, *e.g.*, a decrease in CPU utilization after a configuration change aimed to increase efficiency, or show the unexpected impact occurrences, *e.g.*, a sudden increase in page view response delay after a software upgrade.

The assessment of any *single* software change should be in a low-computational way, for the following two reasons. (1) Hundreds to thousands of KPIs should be detected after a single software change; (2) The operations team should assess tens of thousands of *software changes* every day.

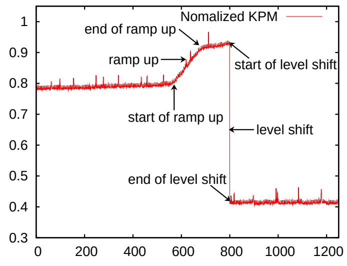

Figure 2: Examples of level shift and ramp up/down

3. FUNNEL DESIGN

Recall that our aim is to provide an automatic tool for rapid and robust impact assessment of software changes. To achieve this goal, we designed FUNNEL, which is composed of two main components, as shown in Fig. 3.

Impact set identification. In the studied search engine company, the operations team names the services based on the service hierarchy. We believe this practice is not uncommon in other companies. FUNNEL derives the relationship among services using the naming rules. Inspired by [18], FUNNEL automatically identifies the impact set, that is, the set of servers, *instances*, and services that may be impacted (§3.1) based on the change deployment logs and the relationships among services,.

Performance change detection and determination. FUNNEL uses an SST based performance change detection method. It improves the robustness of SST (§3.2.2) and introduces matrix compression and implicit inner product calculation to solve the high computational cost of SST (§3.2.3). The improved SST can detect KPI changes rapidly and robustly (step 2 in Fig. 3). FUNNEL then determines whether the changes are caused by software changes. If the KPI is not the KPI of *affected services* (services that are related to the service where the software change is deployed) (step 4 in Fig. 3) *and* if the operations team rolls out the software change using Dark Launching (step 7 in Fig. 3), FUNNEL excludes the impact of other factors (step 9 in Fig. 3) by applying a DiD method (§3.2.4) using KPIs of *cservers* and *cinstances* (step 8 in Fig. 3). Otherwise, based on the DiD method, FUNNEL uses historical measurements of KPIs (step 5 in Fig. 3) to exclude the impact of seasonality (§3.2.5), *i.e.*, the time of day and the day of week effects(step 6 in Fig. 3).

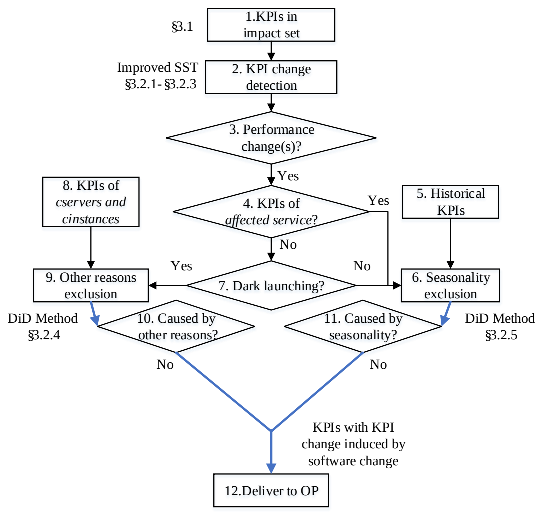
§3.1
1.KPIs in impact set
Improved SST §3.2.1-§3.2.3
2. KPI change detection
3. Performance change(s)?
Yes
8. KPIs of cservers and cinstances
4. KPIs of affected service?
Yes
5. Historical KPIs
No
Yes
No
9. Other reasons exclusion
7. Dark launching?
6. Seasonality exclusion
DiD Method §3.2.4
10. Caused by other reasons?
11. Caused by seasonality?
DiD Method §3.2.5
No
No
KPIs with KPI change induced by software change
12.Deliver to OP

Figure 3: FUNNEL design

3.1 Impact Set Identification

The effect scopes of different types of software changes are different: some are local, *e.g.*, a configuration change aimed at balancing traffic only influences the performance of the servers where the change is conducted, while others are global, *e.g.*, an upgrade in an advertising system can have an impact on almost all types of Internet-based services. A false impact scope of a software change may give rise to delayed detection after a performance degradation or a large number of false alarms.

It is straightforward to identify the impact scope of a software change on servers because only the performance of the servers where the software change is conducted can be directly affected. Therefore, the impact set of servers consists of *tservers*, *i.e.*, the servers on which the software change is deployed. The set of *tservers* is directly obtained from the software change logs.

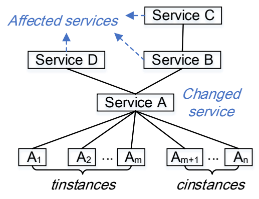

Figure 4: Example of service relationship

As Fig. 4 shows, for a software change deployed on Service A, suppose that Service A has multiple *instances* $(A_1, A_2, ..., A_n)$, Service A is related to Service B and Service D (*i.e.,* Service A send requests and responses to Service B and Service D, and the relationships among services are available to the operations team), and Service B is related to Service C. We refer to service A as the *changed service*, and service B, service C and service D as the *affected services*. For a specific software change deployed on $(A_1, A_2, ..., A_m)$, $(A_1, A_2, ..., A_m)$ are the *tinstances*, and $A_{m+1}, ..., A_n$ are the *cinstances*, which are the service A's *instances* running on the servers on which the change is not deployed yet. The impact set consists of the *tinstances*, the *changed service*, and the *affected services*. We do not include any *instances* of the *affected services* in the impact set because it is unlikely that any *instance* KPI of the *affected services* is individually affected for load balancing reason. Instead, they are more likely to be affected by the same impact. Thus, including their aggregation, *i.e.*, the *affected service* KPI in the impact set, is sufficient for studying the impact of software changes on the *affected services*.

Except for the *affected services*, the elements in the impact set all belong to one service. The operations team usually does not deploy two software changes in one service at the same time based on common practice (to avoid complications). Therefore, except for the *affected services* all the elements in the impact set of a specific software change are not likely to be impacted by other software changes. For the *affected services* in the impact set, if they are the *affected services* or the *changed services* of other software changes, the operations team can manually determine the cause of the behavior changes in the *affected services* when the results are delivered by FUNNEL to them.

In addition, it is possible that two services share the same network infrastructure, such as TOR, aggregation switch, access router. However, based on intuition and the operations team' experience, even if some two services share the same network infrastructure, the KPIs of one service is little impacted by the other service. Therefore, we do not consider the interference of services that share the same network infrastructure.

FUNNEL investigates *all* the KPIs in the impact set automatically. A time-series is constructed for each KPI by dividing the original event series into equal time-bins. One min is used as the time-bin in FUNNEL since our goal is to achieve a rapid impact assessment of software changes.

3.2 Change Detection and Determination

For a given time series of KPIs, our mission is to determine whether behavior changes (level shifts, ramp up/downs) exist and whether they are caused by software changes. Our objective is to provide a rapid and robust KPI change detection and determination tool, which goes beyond traditional change detection methods that compare means, medians, distributions, *etc.*, before and after software changes. For example, the CUSUM approach used in [20] suffers from low accuracy in the face of KPIs with strong seasonality. In addition, its long detection delay also makes the CUSUM method unsuitable for our scenario.

SST, which is based on SVD, has been shown to be accurate and rapid in performance change detection in other research fields [12, 22]. SST projects training data into a normal subspace and finds the difference between the normal subspace and the data that needs to be tested. However, SST suffers from low accuracy in the case of normal subspace contamination. Since the training data may contain outliers as a result of previous software changes, network device breakdowns, network attacks, *etc.*, baseline contamination often occurs in practice. In addition, due to the complexity of SVD, SST suffers from high computational cost.

Our core idea is to improve the robustness and reduce the computational cost of SST, and then combine the improved SST with DiD to determine the impact of software changes. We first provide a brief description of SST's detection of performance changes.

3.2.1 SST: Singular Spectrum Transform

SST [22] is based on SVD of the Hankel matrix [5], and its main idea is to find the difference before and after a point $x(i)$ at time t. Specifically, the algorithm compares a representation of the dynamics of a few points before $x(i)$ and a few points after $x(i)$. The difference is normalized by $x_s(i)$.

The dynamics of the points before $x(i)$ can be denoted by a Hankel matrix:

$$B(t) = [q(t-\delta), ..., q(t-1)] \tag{1}$$

where $q(t) = [x(t-w+1), ..., x(t)]^T$, ω and δ are the size and the number of the overlapping windows respectively .

To find the singular values and vectors of the Hankel Matrix, SVD is used:

$$B(t) = U(t)S(t)V(t)^T \tag{2}$$

where $S(t)$ is the singular value matrix and $S(i-1, i-1) \leq S(i,i) \leq S(i+1, i+1)$, and $U(t)$ and $V(t)^T$ are unitary matrixes. The columns of $U(t)$ are the eigenvectors of $B(t)B(t)^T$. The first η eigenvectors of $U(t)$ ($U_\eta(t)$) are used to denote the past change pattern.

Then, for the future points of $x(i)$, a similar procedure is used to find the largest change in the dynamics by concatenating γ overlapping windows of size ω, starting at ρ points after $x(i)$:

$$A(t) = [r(t+\rho), ..., r(t+\rho+\gamma-1)] \tag{3}$$

where $r(t+\rho) = x(t+\rho), ..., x(t+\rho+\omega-1)^T$.

The eigenvector $\beta(t)$, which represents the direction of the maximum change in the future of the time-series, is:

$$A(t)A(t)^T u^\rho = \mu u^g \tag{4}$$

$$\beta(t) = u^\rho_\gamma \tag{5}$$

where $\gamma = \arg\min_i(\mu_i)$.

Then, the projection of $\beta(t)$ onto U_η is used to quantify the discordance between $\beta(t)$ and U_η:

$$\alpha(t) = \frac{U_\eta(t)^T \beta(t)}{\| U_\eta(t)^T \beta(t) \|} \tag{6}$$

The change score is calculated as the cosine of the angle between $\alpha(t)$ and $\beta(t)$ as:

$$x_s(t) = 1 - \alpha(t)^T \beta(t) \tag{7}$$

The intuition behind the algorithm is that $\beta(t)$ is in or very near to the direction of the maximum change in the past denoted by U_η when there is no change in the time-series.

SST suffers from three problems as follows. (1) Five different parameters (ρ, γ, η, δ, ω) must be specified in SST. Domain knowledge can help find proper values for δ and ω, but fails in choosing the other parameters; (2) SST degrades fast in terms of accuracy when the input time-series includes significant noises [21]; (3) SST is based on SVD which suffers from high computational cost, and thus SST is not computationally efficient and not suitable for KPI change detection when the number of KPIs is large [14].

3.2.2 Improving the Robustness of SST

In this section, we describe our approach to addressing the first two aforementioned problems. Our approach is motivated by the study reported in [21].

As suggested in [14, 21], we set $\rho = 0$, $\gamma = \delta$, and since a value of 3 or 4 is suitable for η even when ω is on the order of 100 empirically, we set $\eta = 3$.

To estimate the change score around every point and alleviate the impact of noises in the time series, we attempt to use more information from the future matrix $A(t)$ than SST does by utilizing the η eigenvectors of $A(t)A(t)^T$ with the smallest corresponding eigenvalues ($\lambda_{1:\eta}$), rather than using only the first one.

Based on Eq. 4, the eigenvector $\beta_i(t)$ is calculated as:

$$\beta_i(t) = u^g_i \tag{8}$$

where $i \leq \eta$, and $\lambda_{j-1} \leq \lambda_j \leq \lambda_{j+1}$ for $1 \leq j \leq w$.

The change score of the point $x(i)$ at time t is calculated:

$$\hat{x}(t) = \frac{\sum_{i=1}^{\eta} \lambda_i \times \varphi_i(t)}{\sum_{i=1}^{\eta} \lambda_i} \tag{9}$$

where

$$\varphi_i(t) = 1 - \sum_{j=1}^{\eta} (\beta_i(t) u_j^T)^2 \tag{10}$$

where u_j is the column of U_η. As described in [14], Eq.10 is consistent with Eq.7.

To alleviate the effect of noise on the final change score, we then filter the sections where the median and the median absolute deviation (MAD) of $\hat{x}(t)$ remain nearly constant. The intuition behind the filtering step is that the accuracy of SST is reduced mainly when the noise takes over the original signal in the time-series. Since the mean and standard deviation for Gaussian distribution are not very robust in the presence of large changes or outliers, we use the median and MAD rather than the mean and standard deviation [21]. The combination of median and MAD has been proved to be a more robust approach even when outliers occur [18]. The change score $\tilde{x}(t)$ is then updated:

$$\begin{aligned} \tilde{x}(t) = \hat{x}(t) \times \mid median_a(t) - median_b(t) \mid \\ \times \mid \sqrt{MAD_a(t)} - \sqrt{MAD_b(t)} \mid \end{aligned} \tag{11}$$

where $median_a(t)$ and $median_b(t)$ are the medians of $x(t)$ in the time-series of length $(2\omega - 1)$ before and after the point $x(i)$ at time t, respectively, and

$$MAD_a(t) = median(x_{ia}(t) - median_a(t)) \tag{12}$$

where $x_{ia}(t)$ is the time-series of length $(2\omega - 1)$ before $x(i)$ at time t. It is similar to $MAD_b(t)$.

3.2.3 Reducing the Computational Cost of SST

In addition to the robustness problem, SST also suffers from high computation cost because of the SVD procedure and thus its direct deployment in our scenario is not feasible. This section summarizes the work in [14]. First, we applied the Implicit Krylov Approximation (IKA) algorithm to reduce the computation cost for SST. The essence of the IKA algorithm is matrix compression and implicit inner product calculation, and the efficiency of the algorithm was demonstrated in [14].

First, let $C = B(t)B(t)^T$, and $\eta < k < \omega, k \in N$. Since it is empirically true that the change score is not very sensitive to δ [14], we set $\delta = w$ as the IKA algorithm requires, and then, we get $\gamma = \delta = \omega$. In addition, suppose that T_k is a k-dimensional tridiagonal matrix; $a_1, ..., a_k$ and

$b_1, ..., b_{k-1}$ are the diagonal and subdiagonal elements of T_k. At each time t, we first compute $\beta_i(t)$, and then run Lanczos $(C, \beta_i(t), k)$ [11] to obtain T_k. Using the QL iteration [23], the eigenvectors of the tridiagnal matrix T_k can be calculated extremely fast. Based on the η top eigenvectors $x_1, ..., x_l$ of T_k, we can obtain $\varphi_i(t)$ as:

$$\varphi_i(t) \simeq 1 - \sum_{j=1}^{\eta} {x_j}^2 \tag{13}$$

Then the change score can be calculated by Eqs.9 and 11.

Based on [14], the dimension of the Krylov subspace k can be set as:

$$k = \begin{cases} 2\eta, \eta \in even \\ 2\eta - 1, \eta \in odd \end{cases} \tag{14}$$

For a service that needs quick mitigation on false software changes, ω can be set to a small value such as 5. For a service that needs more precise assessment of software changes, ω can be set to a lager value such as 15.

3.2.4 *Excluding Other Reasons for Non-affected-service KPIs in Dark Launching*

In addition to software changes, the KPI changes can also be induced by other factors including seasonality, network attacks, *etc.* The impact of these factors can over-shadow the impact assessment of software changes.

To solve this problem, motivated by [15, 19], we apply a split testing method to compare the relative performance of the treated group (KPIs of servers/services/*instances* in the impact set) and the control group (KPIs of *cservers/cinstances*). The intuition behind the comparison is that other factors except software changes influence both the treated group and the control group.

Treated group and control group. In Dark Launching, the operations team first deploys the software change on a subset of servers and *instances*, and then, rolls it out to all servers and *instances* that belong to the same service. It is straightforward to identify the control group and the treated group of servers based on the software change logs. The KPIs of *tservers* constitute the treated group of servers, while the KPIs of *cservers*, *i.e.*, servers that belong to the same service as *tservers* but without software changes, constitute the control group of servers. Since the KPI of the *changed service* is an aggregation of the KPIs of the *tinstances*, determining the relative performance of the *tinstances* is sufficient: if no performance changes in *tinstances* are caused by software changes, it is not necessary to study the impact on the *changed service*. Therefore, the treated group of service consists of the KPIs of *tinstances*, while the control group of services is constituted of KPIs of *cinstances*, *i.e.*, instances that belong to the same service as *tinstances* but without software changes. This is based on the four following observations about Internet-based services. (1) *Instances* and servers that belong to the same service exhibit high-level spatial correlation or statistic dependency because of the similarity in traffic load, memory usage, etc., thanks to load balancing; (2) It is very likely that any non-software change factors will introduce similar performance impact on all servers and instances of the same service; (3) A performance change induced by the software change introduces a relative difference in performance between the treated group and the control group; (4) As studied in [7], only a small fraction (less than 3%) of edge links are hotspots in datacenter networks. In other words, a small fraction of servers are hotspots. Even though there are KPIs of hotspot servers in the control group, most of the KPIs in the control group are not of hotspot servers. We use the *average* of all of the KPIs in the control group to eliminate performance changes caused by other factors, and the large number of KPIs in the control group can alleviate the impact of hotspots.

The operations team's common practice is not to deploy two software changes in a specific service at the same time. Furthermore, the KPIs of a specific control group all belong to a specific service. Therefore, KPIs in the control group are unlikely to be impacted by some other software changes.

For each *affected service*, if it is influenced by software changes, all of its *instances* will be affected. Therefore, there are no *cinstances* for *affected services*. We exclude the "Full Launching" and other reasons for *affected services* in §3.2.5.

DiD method. DiD is one of the most popular tools in econometrics [26] and health care [27] for evaluating the effects of interventions that are instituted at a particular point in time. In this study, DiD is used to compare changes over time in the treated group with those over time in the control group, and to attribute the difference-in-differences to the effects of software changes. DiD is based on the assumption that, in the absence of software changes, the difference between the average KPIs for the treated group and those for the control group remains stable over time. Although [19] argued that DiD suffered from low accuracy in impact assessment of changes in cellular networks, it turns out to perform quite well in our scenario because the treated group and the control group exactly follow the above assumption.

The basic DiD framework can be described as follows. Let $Y(i,t)$ be the KPI i at time t. The performance is observed in a pre-software-change period $t = 0$, with length ω as described in §3.2.1, and in a post-software-change period $t = 1$ with length ω. Some fraction of KPIs are exposed to the software change between these two periods. If KPI i has been exposed to the software change prior to period t, then $D(i,t) = 1$, and $D(i,t) = 0$ otherwise. In other words, those KPIs with $D(i,1) = 1$ are in the treated group, and those KPIs with $D(i,1) = 0$ are in the control group. Obviously, $D(i,0) = 0$ for all i, because the KPIs are exposed to the software change only after the first period.

To obtain the standard errors and significance levels for the DiD estimator, a linear parametric model is used [6]:

$$Y(i,t) = \theta(t) + \alpha \cdot D(i,t) + \xi(i) + \upsilon(i,t) \tag{15}$$

where $\theta(t)$ is a time-specific parameter, α denotes the impact of software changes, and $\xi(i)$ is an KPI-specific parameter. $\upsilon(i,t)$ is a transient shock at each period, *i.e.*, $t = 0, 1$, with mean zero. A sufficient condition for determination using DiD is $P(D(i,1) = 1|\upsilon(i,t)) = P(D(i,1) = 1)$.

Then, the impact estimator of software change, α, is:

$$\begin{aligned}\alpha = \{E[Y(i,1)|D(i,1) = 1] - E[Y(i,1)|D(i,1) = 0]\} \\ -\{E[Y(i,0)|D(i,1) = 1] - E[Y(i,0)|D(i,1) = 0]\}\end{aligned} \quad (16)$$

where $E(\cdot)$ denotes the expectation.

If the KPI changes are caused by factors excluding software changes, then there is no change in the relative performance between the treated group and the control group, thus the DiD impact estimator, α, should be near zero. Therefore, if $\alpha \approx 0$, we consider that the performance changes are not induced by software changes. If $\alpha \gg 0$ or $\alpha \ll 0$, then we consider that there is a relative increase or decrease in the treated group as compared to the control group, and the likelihood that the performance changes are caused by a software change is high.

Empirically, for a service which is sensitive to KPI change, such as advertisement, online shopping, the threshold of α can be set to a small value like 0.5. Otherwise, the threshold can be set larger.

3.2.5 Excluding Other Reasons for Related Services and Full Launching Manner

For *affected services*, there are no *cservers* or *cinstances*. Moreover, if the operations team deploys software changes on all servers at one time (*i.e.*, Full Launching), the control group is also empty. In addition to software changes, seasonality may also give rise to KPI changes [10, 18, 19]. Thus, we need to exclude KPI changes induced by seasonality. For a specific KPI of servers/services/*instances* in the impact set, FUNNEL compares the measurements of KPI around the software change and the KPI measurements in the same period of day but on historical days, since seasonality impacts the KPI measurements around the software change and the historical KPI measurements similarly.

Treated group and control group. For a given KPI of a server/service/*instance* in the impact set, the treated group consists of the measurements of KPIs around the software change, while the control group consists of the historical measurements of KPIs. Specifically, to exclude the performance changes due to the time-of-day or day-of-week pattern and exclude the influence of baseline contamination, we use the measurements of KPIs of 30 days before the day of the software change to construct the control group. This is based on the observation that there is almost no relative performance change between the control group and the treated group if the performance change is induced by seasonality.

FUNNEL also applies DiD method to compare the relative performance between the control group and the treated group. Specifically, $t = 0$ for a pre-software-change period of length ω as described in §3.2.1 in the treated group, and for the same period of day but on historical days in the control group. Similarly, $t = 1$ denotes a post-software-change period, of length ω, in the treated group, and the same period of day but on historical days in the control group.

4. FUNNEL EVALUATION

In this section, we evaluate FUNNEL's performance. We compare FUNNEL with other software change assessment methods including CUSUM [20] and MRLS [18] that have previously been deployed for upgrade assessment in the network infrastructure. We implemented FUNNEL, CUSUM, and MRLS with C++. We use the software change and KPI data from a few real-world Internet-based services offered by a top global search engine. The evaluation using real-world data is challenging because of the lack of a ground truth [19]. We used manual assessment results by the operations team as our ground truth for evaluation.

The focus of FUNNEL is to detect KPI behavior changes in a *rapid* and *robust* way, which is achieved by the improved SST. The introduction of DiD method helps to determine whether the KPI changes are caused by software changes, by comparing the performance between the control/treated group. We decided not to compare FUNNEL with Litmus because the robust regression approach in Litmus [19] is mainly used for inferring the relative performance change between the treated and the control group, and the comparison adds no additional value.

Our results in this section show FUNNEL performs significantly better than CUSUM and MRLS in accuracy, detection delay, and computational cost.

4.1 Data Sets

In cooperation with the operations team, we randomly picked 19 moderate-sized services (the manual assessment efforts for large services would be prohibitive) over a 2-day period, then we got 6277 software changes in total. We ran the algorithm in §3.1 to identify the impact set, and collected the KPI measurements of *tservers/tinstances/changed* services/*affected* services and the KPI measurements of *cservers/ cinstances*. Note that running this algorithm is equally beneficial to FUNNEL, CUSUM and MRLS, and is not biased towards FUNNEL. The details of manual inspection are as follows. For a given software change, first we aggregated the KPIs of *tinstances*/*tservers* by calculating the average measurements of *tinstances*/*tservers*. The operations team then manually inspected whether behavior changes occurred in the KPIs of the *changed* service, the *affected* services, and the *aggregation* KPIs of the *tinstances* and the *tservers* around the time of the software change. The operations team found that in total 83 software changes had behavior changes in the KPIs of *changed* service/*affected* service or *aggregation* KPIs of *tservers*/*tinstances* shortly after each of these software changes. For each behavior change, the operations team then manually inspected the control group and determined whether it is actually caused by software change. Eventually, the operations team selected 72 out of 6277 software changes that induced KPI changes. We admit that behavior changes could occur in one or more *individual* KPIs of the *tservers/tinstances* in the 6194 (6277 - 83) software changes. However, manually inspecting all the KPIs of the 6194 software changes, *i.e.,* about 600 thousands KPIs, is a huge amount of work. Alternatively, we *randomly* selected 72 out of the remaining 6194 software changes. For all selected 72 software changes, the operations team manually carried out the investigation, and found that there were no KPI changes induced by the software changes. We use the 72 software changes that induced changes in KPIs and the 72 selected software changes that did not for the evaluation, and the manual assessment results of the 144 (72+72) software changes served as the ground truth.

The data used in our evaluation exhibited different characteristics including seasonality, variability, stationarity, and different levels of baseline contaminations, which provided a relatively exhaustive validation.

Generally, the CPU context switch count of servers varies frequently, while the memory utilization remains stationary. In addition, both the CPU context switch count and the memory utilization indicate the health status of servers. Specifically, the CPU context switch count indicates the computational efficiency and the number of threads after software changes, while the memory utilization indicates whether a software change introduces memory leaking. Thus we used the CPU context switch count and the memory utilization as the KPIs of all the servers in the evaluation. The KPIs of a given service/*instance* are defined by the operations team and they differ from one service/*instance* to another service/*instance*.

We collectively refer to the combination (S_i, c_i, k_i) as an *item*, where S_i denotes a software change, c_i is a server, service, or *instance* in the impact set, and k_i is a KPI of c_i.

A total of 9982 *items* were included in the evaluation, to which 144 software changes (described earlier in this section), and 931 servers were related. More specifically, the k_i of 931, 931, 8120 *items* were CPU context switch count, memory utilization, KPIs of services/*instances*, respectively. In addition, based on the operations team's assessment, there were 968 *items* that had been labelled as having performance changes introduced by software changes in the 72 software changes that induced behavior changes.

Based on empirical experience, if a software change in an Internet-based service has a negative impact, the KPIs usually change shortly after the software change. The operators think that 1 hour is enough for software change assessment. Among the 144 software changes, 108 were deployed with Dark Launching, while the remaining 26 were not. If a software change was conducted with Dark Launching, and the KPIs were not of *affected services*, we will construct the treated group using the KPIs of *tservers/tinstances* 1 h before and after the software change, and construct the control group using the KPIs of *cservers/cinstances* in the same period. Otherwise, the treated group will consist of the measurements of KPIs 1 h before and after the software change, and the control group will consist of the measurements of KPIs in the same period but on historical days (30 days).

We constructed a time-series, $x(1), x(2), ..., x(n)$, for each *item* by dividing the original measurements of KPIs into equal time-bins of 1 min. Each method took a *time window* of $x(i), x(i+1), ..., x(i+W)$ as its input to construct a matrix (FUNNEL, MRLS) or calculate a cumulative sum (CUSUM). For a fair comparison of the accuracy among FUNNEL, CUSUM and MRLS, the length of the sliding input *time window* W for each method was set as the one that achieved the best accuracy ($W_{FUNNEL} = 34$ (*i.e.*, ω in §3.2.3 is set 9), $W_{MRLS} = 32, W_{CUSUM} = 60$ in our scenario). The time window moves forward every minute. For example, FUNNEL first detects and determines performance changes for $x(1), x(2), ..., x(34)$, and then for $x(2), x(3), ..., x(35)$, *etc.* Note that the values of other parameters of CUSUM, MRLS and FUNNEL (α) are also set to the best for the corresponding algorithm's accuracy. We believe the above method draws the same conclusion as the method that changing the value of the parameters, calculating the accuracies and plotting the receiver operating characteristic (ROC) curves.

Empirically we set a threshold of 7 minutes in FUNNEL to declare a change in a time series as a level-shift or ramp-up/down rather than a one-off event.

4.2 Comparison of Accuracy

We now compare the accuracy of FUNNEL, the improved SST without DiD, CUSUM, and MRLS. For each *item*, based on the ground truth provided by the operations team, we knew the outcome - either having the KPI changes induced by software changes or not. For each method, we labelled its outcome as true positive (TP), true negative (TN), false positive (FP), and false negative (FN). True positives were *items* with KPI changes caused by software changes that were accurately determined as such by the method, and true negatives were *items* that were accurately determined as having no KPI changes induced by software changes. If the method determined a KPI change caused by a software change while there was no KPI change or the KPI change was not induced by a software change, we then labelled the *item* as a false positive. False negatives were KPI changes induced by a software change that were incorrectly missed by the method. We calculated Precision, Recall, true negative rate (TNR), and Accuracy as: Precision $=\frac{TP}{TP+FP}$, Recall $=\frac{TP}{TP+FN}$, TNR $=\frac{TN}{TN+FP}$, Accuracy $=\frac{TP+TN}{TP+TN+FP+FN}$ [19].

4.2.1 Comparison Results

Based on the characteristics of KPIs, the 9982 *items* were divided into three types: seasonal, stationary, and variable [18]. It is clear that the *items* with CPU context switch count were variable, and the *items* with memory utilization were stationary. If the KPI of a given *item* has strong seasonality, then the *item* is seasonal. Similarly, the *item* with KPI of strong variability is variable, and the rest are stationary. For *items* of service, 705 *items* were seasonal, 2702 *items* were stationary, and 4713 *items* were variable.

To examine the performance of FUNNEL in handling sea-

Table 1: The Precision, Recall, TNR and Accuracy of seasonal, stationary and variable data for FUNNEL, Improved SST, CUSUM and MRLS

Algorithm	Type	Total	Precision	Recall	TNR	Accuracy
FUNNEL	Seasonal	28500	98.28%	100.00%	100.00%	100.00%
	Stationary	129943	100.00%	100.00%	100.00%	100.00%
	Variable	215339	68.47%	99.48%	99.88%	99.88%
Improved SST	Seasonal	28500	1.10%	100.00%	81.93%	81.96%
	Stationary	129943	14.28%	100.00%	98.44%	98.44%
	Variable	215339	15.04%	99.48%	98.50%	98.50%
CUSUM	Seasonal	28500	0.76%	84.21%	77.97%	77.98%
	Stationary	129943	10.34%	98.52%	97.78%	97.78%
	Variable	215339	17.92%	96.34%	98.82%	98.81%
MRLS	Seasonal	28500	100.00%	87.72%	100.00%	99.98%
	Stationary	129943	9.23%	97.33%	97.51%	97.51%
	Variable	215339	0.61%	97.04%	57.85%	57.95%

sonal and variable KPIs, we reorganized the above KPI items based on the KPI types, and then compared different methods. For each method, we *multiplied* the TPs, TNs, FPs, and FNs of the 72 software changes that did not introduce behavior changes with 86 (6194/72), and added the result to the TPs, TNs, FPs, and FNs of the 72 software changes which induced KPI changes. We believe that this provides a reasonable approximation of real TPs, TNs, FPs, and FNs. Eventually, we calculated the accuracy, recall, precision and TNR based on the *synthetic* true positives, true negatives, false positives, and false negatives. Table 1 shows the aggregated results for each method and each KPI type. FUNNEL performed the best across all three kinds of KPIs. The improved SST, CUSUM and MRLS performed well in the face of stationary KPIs. However, because CUSUM and the improved SST failed to exclude the impact of seasonality, they detected performance changes induced by software changes with low accuracy when the KPIs had strong seasonality. MRLS was sensitive to spikes, and it was hardly feasible to modify MRLS to detect level shifts or ramp up/downs only. Thus a large portion of the *items* with variable KPIs were incorrectly determined as having KPI changes.

4.3 Comparison of Computational Cost

Since millions of KPIs should be monitored to determine whether they are impacted by software changes, it is important that the computational cost of the impact assessment be relatively low for the sake of scalability and deployability. In this section, we compare the computational cost of FUNNEL, CUSUM, and MRLS.

Motivated by the evaluation method presented in [28], all three methods were deployed on the same server (CPU information: 12 Intel(R) Xeon(R) CPU E5645 @ 2.40GHz) with a single thread. The CPU utilization remained 100% while the process of each method was running so that we could use the total time to evaluate complexity [28].

Table 2 shows the average computational time taken by FUNNEL, CUSUM and MRLS to detect the KPI changes in a single *time window*. With the introduction of matrix compression and implicit inner product calculation, FUNNEL was highly computational efficient. Specifically, compared with MRLS, FUNNEL was more than 7000 times faster in computational speed. Moreover, FUNNEL reduced 77.42% of computational cost as compared to CUSUM. Suppose that one million KPIs need to be monitored and determined for the impact assessment of software changes, the KPIs are collected and detected every minute, and the implementation runs on the same types of CPU. As the last row in Table 2 shows, if we apply MRLS as the software change assessment method, we need at least 47526 cores, *i.e.*, 3960 servers if each server has 12 cores. CUSUM needs 31 cores and 3 servers. However, with FUNNEL, one server is fully capable of detecting and determining all one million KPIs. FUNNEL is thus quite competent to assess the impact of software changes online in large Internet-based services.

Table 2: Comparison of computational time

Method	FUNNEL	CUSUM	MRLS
Run time per *time window*	401.8 μs	1.846 ms	2.852 s
# Cores for one million KPIs	7	31	47526

4.4 Comparison of Detection Delay

When detecting level shifts, and ramp up/downs, all detection methods need some data points (thus time) to finish. However, rapid detection of performance changes and determination of software change impacts is quite necessary for timely damage mitigation for Internet-based services. In this section, we compare the detection delay of different methods and show that FUNNEL has a much lower detection delay, thanks to the use of SST.

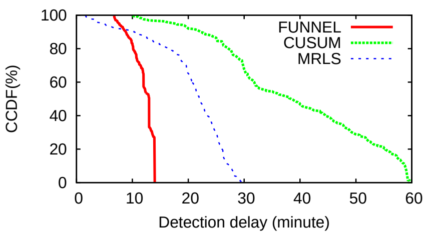

Figure 5: CCDFs of detection delay

For each *item*, the operations team labelled the start of KPI changes (if any, as shown in Fig. 2), which served as the ground truth information for comparing different methods. We defined the detection delay as the time between the start of a KPI change and its detection by a method. Suppose that a method correctly detects and determines a KPI change firstly when the input *time window* is $x(i+1), x(i+2), ..., x(c), ..., x(i+w)$, and the KPI change starts at time c, then the detection delay is $(w-c)$ minutes. Note that the detection delay defined above does not include the delay due to the computational cost (previously evaluated separately in §4.3). During evaluation, each method is given sufficient CPU power to finish processing one-window's worth of data within one time window.

Fig. 5 shows the Complementary Cumulative Distribution Functions (CCDFs) for the detection delay of FUNNEL, CUSUM, and MRLS. The median delays were $M_{MRLS} = 21.3min$, $M_{FUNNEL} = 13.2min$, and $M_{CUSUM} = 37.7min$. FUNNEL reduced 38.02% of detection delay compared with MRLS, and 64.99% for CUSUM.

As aforementioned, in FUNNEL we set a threshold of 7 minutes to declare a change in a time series as a level-shift or ramp-up/down rather than a one-off event. Occasionally, MRLS can detect a level shift within 7 minutes, at the cost of much more false positives. In these cases, FUNNEL is slower than MRLS as shown in the top left corner of Fig. 5

The distribution of the detection delay for FUNNEL was more concentrated and the longest detection delay of FUNNEL was much shorter than that of CUSUM and MRLS. This gives FUNNEL a very significant advantage, because a software change may introduce a great loss due to impairment if an unexpected impact of a software change causes poor user experience or degraded advertising income and is detected only after a long delay. Thus, in terms of detection delay, overall FUNNEL is more suitable than CUSUM and MRLS for online impact assessment of software changes in large Internet-based services.

5. OPERATIONAL EXPERIENCE

We deploy the multi-threaded FUNNEL prototype on one server with a 12-core Intel(R) Xeon(R) CPU E5645 @ 2.40 GHz. FUNNEL accesses the software changes of a few dozens of services offered by the search engine company. Table 3 shows some daily statistics for a specific one-week period which we studied in details in this section.

Ideally, we would like to measure the Accuracy, Precision, Recall and TNR as in §4.2 for this one-week period. However, it is prohibitive for the operations team to label more than 2 million KPIs of *all* services daily. Therefore, we made a compromise, and only asked the operations team to verify the KPI changes detected by FUNNEL (10 thousands per day, 4th column in Table 3). This allows us to calculate Precision $= \frac{TP}{TP+FP}$ (in the last column in Table 3).

Based on the operational experience of FUNNEL deployment, we show two representative cases which highlight FUNNEL's detection robustness and speed.

Table 3: Statistics about the implementation of FUNNEL

#software changes	# software changes that have impact	#KPIs	#KPI changes	Precision
24119	268	2256390	10249	98.21%

In both two cases, FUNNEL successfully detected performance changes and determined the impact of software changes in a rapid and robust fashion. We applied the prototype of FUNNEL to validate expected performance changes that were direct outcomes of a configuration change in the first case, and to detect unexpected performance changes induced by a software upgrade in the second.

5.1 Robustness to Variable KPIs

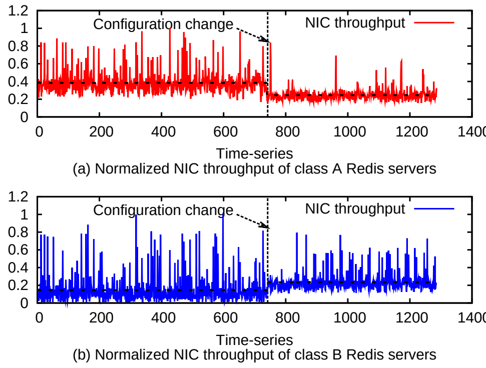

Figure 6: KPI changes induced by a configuration change in the Redis query service

Redis is an advanced data structure store which is used as database, cache and message broker [3]. After a configuration change in the Redis query service, FUNNEL determined that 16 out of 118 KPIs in the impact set had behavior changes which were caused by this configuration change. More specifically, FUNNEL found that immediately after the configuration change, some Redis servers (class A Redis servers) witnessed a negative level shift in NIC throughput as Fig. 6 (a) shows, while in the NIC throughput of other Redis servers (class B Redis servers) a positive level shift occurred, as shown in Fig. 6 (b). Please notice that the X-axis in Fig. 6 is 1 minute, as well as Fig. 7.

The operations team verified that the configuration change was aimed at load balancing and achieved the expected result. Specifically, the Redis query tasks were assigned to class A Redis servers firstly. Unless the class A Redis servers reached the limit of NIC bandwidth capacity, the query tasks would not be allocated to class B Redis servers. This led to a situation where the NICs of class A Redis servers were always busy, while the utilization of the NICs of class B servers was low, which degraded the performance of the Redis query service and shortened the life of NICs in class A Redis servers. The operations team launched a configuration change to balance the traffic between class A and class B Redis servers. As Fig. 6 shows, the configuration change successfully balanced the traffic and had the expected effect.

This case study shows that, although the NIC throughput had strong variability by its nature, FUNNEL still successfully detected and determined the KPI changes induced by the configuration change, demonstrating FUNNEL's capability to handle variable KPIs.

5.2 Rapidly Detecting Unexpected Behavior Changes in Seasonal KPIs

To see how FUNNEL speeds up the detections of unexpected behavior changes, we randomly picked a small fraction of software changes for which FUNNEL does not directly deliver the detection results to the operations team. Instead, the operations team independently assesses software changes without the help of FUNNEL. In the below case, we compare the speed of FUNNEL and manual inspection, and demonstrate the performance of FUNNEL in detecting seasonal KPIs' unexpected changes

During the deployment of FUNNEL, the operations team made a software upgrade aimed at improving the performance of the advertising system. Advertising system is a very large and complex system [8], and in fact 36752 KPIs are included in the impact set in the software upgrade according to FUNNEL. Manually investigating this upgrade's impact is infeasible. 10 minutes after the software upgrade, FUNNEL detected 1141 KPI changes induced by the software upgrade. More specifically, as Fig. 7 shows, the normalized number of effective clicks on advertisements, *i.e.*, clicks on advertisements that are considered as human behavior by the anti-cheating system, decreased dramatically immediately after the software upgrade was conducted.

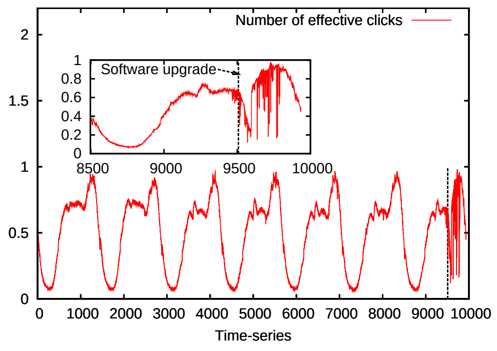

Figure 7: Normalized number of effective clicks

Without using FUNNEL, the operations team independently found the performance change after 1.5 h and then quickly fixed the issue. The operations team confirmed that the occurrence of the unexpected behavior changes due to the software upgrade was a real significant incidents. It was later carefully investigated. Specifically, the anti-cheating service injects a check program, which is implemented with JSON, into every advertisement to determine whether a click on the advertisement is performed by a human or is a cheat executed by an automated program. However, the software upgrade failed to load the JSON program on the iPhone browsers, *i.e.*, the check program was not effective for iPhone users. Therefore, all clicks on the advertisements by iPhone users were considered as cheats, which resulted in the decline in advertising revenue. The number of cheating clicks monitored by the anti-cheating service returned to the normal level after the operations team had remedied the situation, and thus, a positive level shift occurred 1.5 h after the software upgrade.

Had the operations team used FUNNEL to assess the impact of the erroneous software upgrade, they could have much more timely mitigated the loss caused by the upgrade. The KPIs had strong seasonality, but FUNNEL still accurately detected the performance changes and identified that the performance changes are induced by the software upgrade, which demonstrated that FUNNEL performed very well in the face of seasonal KPIs.

6. RELATED WORK

The impact of software changes has attracted considerable attention in recent years [9, 18, 19, 20, 24, 25]. Mahjmkar *et al.* [20] developed MERCURY to detect the performance impact of upgrades in large operational networks. MERCURY uses the CUSUM method to detect behavior changes in KPIs, and applies statistical rule mining and network configuration to identify commonality across the behavior changes. PRISM [18] is developed to reduce the detection delay between the behavior change and the detection. The MRLS method was developed in PRISM to rapidly and robustly detect maintenance-induced behavior changes. However, the MRLS method suffers from high computational cost, and thus is not appropriate in our scenario. Litmus was developed to address the assessment of changes in cellular networks where external factors may over-shadow the assessment [19]. Litmus applies a spatial regression algorithm for the comparison of the treated and the control group.

The detection of behavior changes has a very rich literature. In [16], Principal Component Analysis (PCA) was applied for anomaly detection in network. Yamada *et al.* proposed a change point detection method, additive Hilbert-Schmidt Independence Criterion (aHSIC), which was based on supervised learning, and used the weighted sum of the SIC scores for incorporating feature selection [28]. To detect changes in seasonal time-series, Chef *et al.* applied a time series decomposition based method, week-over-week [10].

7. CONCLUSION

We designed and implemented a new tool, FUNNEL, for rapidly and robustly assessing the impact of software changes in large Internet-based services. For each software change, FUNNEL analyzes all the services, processes, and servers that may be influenced and automatically constructs the impact set. FUNNEL then detects performance changes in the impact set rapidly and robustly by improving the robustness and reducing the computational cost of SST, and determines performance changes induced by software changes using a DiD method. We evaluated FUNNEL by comparing it with CUSUM and MRLS using 144 software changes and showed that FUNNEL achieved high accuracy with short detection delay and low computational cost. Operational experiences of FUNNEL deployment show that FUNNEL can assess the impact of software changes rapidly and robustly in the face of seasonary and variable data. Compared to manual efforts, in one specific case FUNNEL shortens the assessment from 1.5 hours to 10 minutes, saving both time and money.

8. ACKNOWLEDGMENTS

We thank our shepherd Gianluca Iannaccone and the anonymous reviewers for their insightful comments and feedbacks on the paper. We are grateful to Bo Wang, Bolin Yang, Wenhui Xie, Ming Hua, Wei Huang, Xiaoguang Miao and Jiantao Chi, for their invaluable help at various stages of this project. We strongly appreciate Yousef Azzabi, Dapeng Liu, Juexing Liao, Beibei Miao, Yongqian Sun and Xiaohui Nie for their helpful suggestions and elaborative proofreading.

The work was supported by National Natural Science Foundation of China (NSFC) under Grant No.61472214 and No. 61402257, the National Key Basic Research Program of China (973 program) under Grant No.2013CB329105, the Tsinghua University Self-determined Project under Grant No. 2014z-21051,the State Key Program of National Science of China under Grant No.61233007, the National Natural Science Foundation of China (NSFC) under Grant No.61472210, the National High Technology Development Program of China (863 program) under Grant No.2013AA013302.

9. REFERENCES

[1] Back up and running. https://blogs.dropbox.com/dropbox/2014/01/back-up-and-running/.

[2] Facebook chat. https://www.facebook.com/note.php?note_id=14218138919.

[3] Redis. http://redis.io/.

[4] Google apps incident report, gmail partial outage. Technical report, The Google Apps Team, December 2012.

[5] N. Aoki. *State space modeling of time series*. Cambridge Univ Press, 1990.

[6] O. C. Ashenfelter and D. Card. Using the longitudinal structure of earnings to estimate the effect of training programs. *The Review of Economics and Statistics*, 67(4):648–660, August 1985.

[7] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *ACM IMC*, Melbourne, Australia, November 2010.

[8] R. Bhagwan, R. Kumar, R. Ramjee, G. Varghese, S. Mohapatra, H. Manoharan, and P. Shah. Adtributor: Revenue debugging in advertising systems. In *NSDI*, Seattle, WA, April 2014.

[9] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford. A nice way to test openflow applications. In *NSDI*, San Jose, CA, April 2012.

[10] Y. Chen, R. Mahajan, B. Sridharan, and Z.-L. Zhang. A provider-side view of web search response time. *SIGCOMM*, August 2013.

[11] G. H. Golub and C. F. Van Loan. *Matrix computaions*, volume 3. Jhons Hopkins University Press, Baltimore, MD, 2012.

[12] H. Hassani, S. Heravi, and A. Zhigljavsky. Forecasting european industrial production with singular spectrum analysis. *International journal of forecasting*, 25(1):103–118, 2009.

[13] T. Idé and K. Inoue. Knowledge discovery from heterogeneous dynamic systems using change-point correlations. In *SDM*, pages 571–575, Newport Beach, CA, USA, April 2005.

[14] T. Idé and K. Tsuda. Change-point detection using krylov subspace learing. In *SDM*, Minneapolis, Minnesota, April 2007.

[15] R. Kohavi, A. Deng, B. Frasca, R. Longbotham, T. Walker, and Y. Xu. Trustworthy online controlled experiments: Five puzzling outcomes explained. In *SIGKDD*, Beijing, China, August 2012.

[16] A. Lakhina, M. Crovella, and C. Diot. Mining anomalies using traffic feature distributions. In *SIGCOMM*, Philadelphia, PA, USA, August 2005.

[17] Z. Lin, M. Chen, and Y. Ma. The augmented lagrange multiplier method for exact recovery of corrupted low-rank matrices. *arXiv preprint arXiv:1009.5055*, 2010.

[18] A. Mahimkar, Z. Ge, J. Wang, J. Yates, Y. Zhang, J. Emmons, B. Huntley, and M. Stockert. Rapid detection of maintenance induced changes in service performance. In *CoNEXT*, Tokyo, Japan, December 2011.

[19] A. Mahimkar, Z. Ge, J. Yates, C. Hristov, V. Cordaro, S. Smith, J. Xu, and M. Stockert. Robust assessment of changes in cellular networks. In *CoNEXT*, Santa Barbara, California, USA, December 2013.

[20] A. Mahimkar, H. H. Song, Z. Ge, A. Shaikh, J. Wang, J. Yates, Y. Zhang, and J. Emmons. Detecting the performance impact of upgrades in large operational networks. In *SIGCOMM*, New Delhi, India, August 2010.

[21] Y. Mohammad and T. Nishida. Robust singular spectrum transform. In *Next-Generation Applied Intelligence Lecture Notes in Computer Science*, pages 123–132, Tainan, Taiwan, June 2009. Springer.

[22] V. Moskvina and A. Zhigljavsky. Change-point detection algorithm based on the singular-spectrum analysis. *Communication in Statistics: Simulation and Computation*, 32(2):319–352, 2003.

[23] W. H. Press. *Numerical recipes 3rd edition: The art of scientific computing*. Cambridge university press, 2007.

[24] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for network update. In *SIGCOMM*, Helsinki, Finland, August 2012.

[25] R. R. Sambasivan, A. X. Zheng, M. De Rosa, E. Krevat, S. Whitman, M. Stroucken, W. Wang, L. Xu, and G. R. Ganger. Diagnosing performance changes by comparing request flows. In *NSDI*, Boston, MA, USA, April 2011.

[26] W. R. Shadish, T. D. Cook, and D. T. Campbell. *Experimental and quasi-experimental designs for generalized causal inference*. Wadsworth Cengage Learning, 2002.

[27] E. A. Stuart, H. A. Huskamp, K. Duckworth, J. Simmons, Z. Song, M. E. Chernew, and C. L. Barry. Using propensity scores in difference-in-differences models to estimate the effects of a policy change. *Health Services and Outcomes Research Methodology*, 14(4):166–182, August 2014.

[28] M. Yamada, A. Kimura, F. Naya, and H. Sawada. Change-point detection with feature selection in high-dimensional time-series data. In *IJCAI*, Beijing, China, August 2013.