Journal of Network and Systems Management (2024) 32:44
https://doi.org/10.1007/s10922-024-09818-8

SampleHST-X: A Point and Collective Anomaly-Aware Trace Sampling Pipeline with Approximate Half Space Trees

Alim Ul Gias[1] · Yicheng Gao[2] · Matthew Sheldon[2] · José A. Perusquía[3] · Owen O'Brien[4] · Giuliano Casale[2]

Received: 24 November 2023 / Revised: 27 February 2024 / Accepted: 13 March 2024

Abstract
The storage requirement for distributed tracing can be reduced significantly by sampling only the anomalous or interesting traces that occur rarely at runtime. In this paper, we introduce an unsupervised sampling pipeline for distributed tracing that ensures high sampling accuracy while reducing the storage requirement. The proposed method, SampleHST-X, extends our recent work SampleHST. It operates based on a budget which limits the percentage of traces to be sampled while adjusting the storage quota of normal and anomalous traces depending on the size of this budget. The sampling process relies on accurately defining clusters of normal and anomalous traces by leveraging the distribution of mass scores, which characterize the probability of observing different traces, obtained from a forest of Half Space Trees (HST). In our experiments, using traces from a cloud data center, SampleHST yields 2.3× to 9.5× better sampling performance. SampleHST-X further extends the SampleHST approach by incorporating a novel class of Half Space Trees, namely Approximate HST, that uses approximate counters to update the mass scores. These counters significantly reduces the space requirement for HST while the sampling performance remains similar. In addition to this extension, SampleHST-X includes a Family of Graph Spectral Distances (FGSD) based trace characterization component, which, in addition to point anomalies, enables it to sample traces with collective anomalies. For such traces, we observe that the SampleHST-X approach can yield 1.2× to 19× better sampling performance.

Keywords Distributed tracing · Microservices · Anomaly detection · Sampling

Extended author information available on the last page of the article

Published online: 16 April 2024

1 Introduction

The distributed tracing approach is tailored primarily to monitor and profile applications built with the microservices architecture [1]. In a microservice ecosystem, as the size of the architecture increases, the volume of the trace data increases correspondingly [2]. In a typical production setup, this volume can be in the order of several terabytes, and only a fraction of these traces helps in troubleshooting. Hence, a storage budget is often applied to discard the majority of the traces [3]. However, this can result in important data to characterize system events. In this paper, we study the problem of how to sample the most interesting traces that will help in future troubleshooting. This entails not only sampling the traces that represent the overall user behavior but also sampling the anomalous ones.

A common industry practice, to accommodate the storage budget, is to leverage uniform sampling [3] also known as *head-based* sampling. This strategy has a lower hit rate of anomalous traces as the sampling decision is taken before analyzing the trace. This could be addressed with a *tail-based* sampling strategy [4]. Such a strategy reasons on the information contained in the trace before taking a sampling decision. Ideally, such a sampling strategy needs to be online and work with streaming data. This requires that we must decide to save or discard a trace on-the-fly at runtime.

In recent years, multiple tail-based sampling approaches have been proposed [3, 5, 6]. However, these works have issues such as high dimensionality of clustering data, batch processing requirement, low amplitude characterization for anomalous traces, and no explicit consideration of the storage budget. These shortcomings are addressed in our recent work SampleHST [7]. When the storage budget is comparatively lower than the expected anomalies, SampleHST focuses on sampling only anomalous traces. When the budget is higher, both the normal and anomalous traces are sampled, with a bias towards anomalous ones. This bias is fair because, among the sampled traces, the bias increases the representation of anomalous traces groups that are rare compared to the normal groups. In other words, such a bias allows representative sampling [3, 5].

SampleHST relies on the Bag of Words (BoW) model [8] to transform a trace, which is essentially a text document, to a count based representation. This BoW-based representation includes the frequency of unique terms in a document, which is inherently useful for detecting point anomalies [9]. By taking such count-based representation as input, SampleHST generates a distribution of the mass values obtained from a forest of a tree-based classifier, namely Half Space Trees (HST) [10]. This mass distribution is subsequently used to perform an online clustering of the traces. The clustering algorithm, we have designed, is based on the mean-shift clustering algorithm family [11, 12]. Once the clustering process is completed, the sampling decision is taken based on the trace-cluster association, *i.e.*, the trace is likely to be sampled if it is associated with a cluster with low mass values because such clusters represent the rare traces.

This paper presents SampleHST-X, which builds on the ideas of SampleHST and incorporates two main extensions. The first extension relates to the

space requirement for the HST. Depending on the depth the of trees, the storage requirement of the HST increases exponentially. Even though a tree node only stores a mass value, which can be stored in an integer, reducing the space requirement allows to use more HST with higher depths. SampleHST-X realizes this with a novel variant of HST, which we name Approximate HST, that leverages approximate counting to record the mass values in the tree nodes [13]. This can bring advantages when the method is deployed in memory constrained environments as the size of the HST is significantly compressed; such scenario arises for example in use cases that apply tail-sampling directly at the edge close to micro-services connected with Internet-of-Things (IoT) devices, where discarding traces can greatly reduce the cost of communication between edge and backend clouds. The second extension is focused on the detection of collective anomalies [9] in traces using the same sampling pipeline. For this purpose, SampleHST-X leverages the Family of Spectral Graph Distances (FGSD) graph model [14] to characterize a trace. Such a graph model produces a count-based representation of the trace, focusing on its underlying structure and communication pattern. This count-based representation can be used with the online clustering method without additional adaptations.

We first evaluate the performance of SampleHST, with data provided by a commercial cloud service operator, comparing the results with a recently proposed approach for point anomalies [3]. For this production dataset, we see that SampleHST yields 2.3× to 9.5× better sampling performance in terms of precision, recall and F1-Score than prior work. When we consider representative sampling in a high budget scenario, we see SampleHST is 1.6× fairer with respect to the Jain fairness index [15]. When we incorporate approximate HST, *i.e.*, use the SampleHST-X approach, we see a similar performance. While SampleHST-X requires less space for HST, it does not affect the sampling accuracy. To test the performance in case of collective anomalies, we use a dataset generated from a local deployment of the Death Star Bench (DSB) microservice suite [16], which is already used in literature to demonstrate the effectiveness of a trace sampler [5]. We compare the performance of SampleHST-X with the trace sampler proposed in [5]. Here, we also observe that SampleHST-X produces a high sampling performance. In comparison, the sampling performance is 1.2× to 19× better across all the budgets.

In summary, the key contributions of this paper are:

- A trace sampling pipeline SampleHST-X, that extends our recent work SampleHST [7], tailored for distributed tracing in a cloud data center.
- An online clustering method, generalizing the mean shift algorithm [11], that considers non-spherical cluster shapes such as hyper-cubes and hyper-rectangles.
- A novel class of HST, namely Approximate HST, that allow memory savings for HST and reduce the overall resource footprint of distributed trace sampling.
- Simultaneous use of different trace models for anomaly detection with the same sampling pipeline, in particular a BoW-based model for point anomalies and FGSD-based model for collective anomalies.

Experiments using real-world data and a dataset from the literature to compare the sampling performance of our proposed approach with a recent tail-based sampling approach [3] indicate the effectiveness of the SampleHST-X as well as its potential for less resource consumption while maintaining high sampling accuracy.

The rest of the paper is organized as follows. In Sect. 2 we demonstrate the motivation for developing SampleHST-X and how to evaluate such sampling methods. Trace representation and anomaly detection is discussed in Sect. 3. The key aspects of the SampleHST method is presented in Sects. 4–6. The evaluation with industry data is presented in Sect. 7. Subsequently, we present how approximate HST is incorporated in SampleHST-X in Sect. 8. The extension for collective anomalies is presented in Sect. 9. We discuss the related work in Sect. 10. The threats to the validity of SampleHST-X is presented in Sect. 11. The paper is concluded with future research directions in Sect. 12.

2 Motivation

Unlike head-based sampling, a tail-based sampling strategy takes the sampling decision after the response is served, *i.e.*, when the entire trace is available [4]. As a result, the information of the traces can be utilized, which can the proportion of interesting traces in the sample. Before evaluating these samplers, we first need to define the interesting traces and the evaluation criteria. In this section, we first provide this definition and criteria. This is followed by a demonstration showing the limitations of the current samplers that motivates us to the development of SampleHST-X.

2.1 Trace Anomaly Definition

A key point before designing a tail-based sampling strategy is to define what makes a trace more interesting or anomalous. For the rest of the paper, we use the term anomalous traces to indicate interesting traces. In the context of this work, we use the broad definition of anomalies provided in [9]. Precisely, we consider two types of anomalies—point anomalies and collective anomalies.

Point anomalies involve a trace with an abnormally low or high value for one or more trace properties. For example, suppose we define a trace property that counts the occurrence of HTTP code 203 among its spans. If there is a large deviation than the mean value, it can be considered an anomaly. On the other hand, collective anomalies are those where there is an abnormal workflow pattern within traces. For example, if a group of span always creates the same call-chain (like span A calls span B and span B calls span C) and in a particular trace we see a deviation, this trace can be considered an anomaly.

2.2 Evaluation Criteria

It is important to set appropriate criteria for evaluating the performance of a tail-based sampler. In some recent works, researchers have used the notion of

representative sampling to evaluate their proposed method [3, 5]. Although, this adoption of representative sampling provides a visualization of sampling fairness, it is not a numeric metric that can be used for comparison. Further, it does not provide the accuracy of the method, with respect to the storage budget and the fraction of anomalous traces in the data. For example, if the budget (0–1) is smaller than the expected fraction (0–1) of the anomalous or interesting traces, rather than representative sampling, the focus should be on sampling the anomalous traces. This is because, with such a small budget the goal should be to identify the faults in the system, which will appear in the anomalous traces. Representative sampling will increase the number of normal traces being sampled, which is not ideal in this case.

Considering this, we select the evaluation metric depending on the ratio of the anomalies versus the storage budget:

- For infrequent anomalous traces, where the prevalence of anomalies is less than the storage budget, the primary evaluation metric should be the Recall.
- For low storage budgets, where the prevalence of anomalies is greater than the storage budget, the primary evaluation metric should be the Precision.
- When sampling N traces from a collection of traces containing N anomalies, the evaluation metric is the F1-Score.

Here, the definition of precision and recall is same as the definition in the machine learning domain. The F1-Score is the harmonic mean of precision and recall.

2.3 Demonstration

We now demonstrate our motivation for a new tail-based sampler. For this demonstration, we have chosen two recently proposed samplers, one for the point anomalies and the other for the collective anomalies scenario.

For the point anomalies scenario, we use a method [3] based on a hierarchical clustering algorithm, namely Purity Enhancing Rotations for Cluster Hierarchies (PERCH) [17]. In short, this PERCH-based method maintains a fixed size balanced binary tree of the traces and samples from the tree periodically. The tree is maintained in such a way that the frequent traces are more likely to be removed when there is a space shortage. For the collective anomalies scenario, we use the Sifter approach [5]. It considers a trace as a directed acyclic graph (DAG) and predicts the anomalies by inspecting all the k-length paths from the DAG. Here, each path is represented as a n-gram i.e., a sequence of n words. The association among these words are learnt using a variant of *word2vec* algorithm [18]. The traces are sampled based on the familiarity of these n-grams. We have implemented both these approaches using python libraries[1][2].

We now consider a case with production data from a cloud data center. The data contains $\sim$ 5% point anomalies. We provide a storage budget of 5% to PERCH-based

[1] PERCH - https://github.com/iesl/xcluster

[2] Gensim - https://radimrehurek.com/gensim/models/word2vec.html

method and executed it. As mentioned before, since the budget and the percentage of anomalies are roughly the same, the evaluation criteria should be F1-Score. After the execution we observe an F1-Score of 0.1, just slightly above the value of random sampling 0.05. Although, for a low budget of 0.5%, where the evaluation criteria is precision, we see the precision value of the method is 0.41. Under random sampling, this is equal to 0.04. This shows that the PERCH-based method is significantly more applicable in low budget scenarios than moderate budget ones, which underscores the motivation for a sampling technique that works with a wide variety of budgets.

The primary drawback of such clustering method is it is hard to do clustering in high number of dimensions [19], which can be often the case for point anomalies scenario. Thus, for clustering, we need to find a representation that uses less number of dimensions. In addition, this PERCH-based method is a batch process as it needs to maintain a binary tree. This is a major problem when we have no mechanism for temporary storage and need to immediately take the sampling decision.

The production data we have do not contain any collective anomalies. Thus, we use the Death Star Bench (DSB) traces used in [5]. We consider the *Compose Post* API for this experiment. The trace repository[3] contains both the traces with normal and broken executions, where the broken executions are the ones with runtime faults. Following the approach in [5], we consider the broken traces as anomalies and created dataset where 5% of the traces are broken *i.e.,* anomalous. We than execute the Sifter method with 5% budget and obtain an F1-Score of 0.35, which is better than the point anomalies case, but has scope for improvement.

We attribute the limited performance of Sifter to its probabilistic approach of sampling. If the sampling probabilities for the anomalies are not large enough, the trace may not be sampled. To be certain that our implementation is not producing small probabilities, we have checked the probabilities reported for Sifter [5]. The highest probability obtained for Compose Post API is ~ 0.3, which is confirmed from our experiments. This mean that around two out of three anomalous traces might not be sampled. In addition, trace transformation in Sifter is potentially a slow process for large traces, as it checks every path of the corresponding DAG. If we consider data with runtime faults, the normal and erroneous executions will be different and so will be their traces. Thus, rather than checking every path, we can consider the overall DAG structure of trace, reducing the time for trace transformation.

3 Anomaly Detection from Trace Stream

The first step of building a tail-based sampler is to be able to differentiate between normal and anomalous traces. Subsequently, we need to decide how to use that information to sample the traces. In this section, we discuss the first step that is differentiating between the normal and anomalous traces. Here we have used the production trace data containing point anomalies. We present how we can represent these traces and evaluate the performance of different state of the art anomaly detection methods using the representation. Finally, we discuss the adjustments we made

[3] Death Star Bench Traces https://gitlab.mpi-sws.org/cld/trace-datasets/deathstarbench_traces

to Half Space Trees [10], the method that performed best, to better suit our case of distributed tracing.

3.1 Bag of Words Model for Point Anomalies

The production data that we consider consists of spans in the scale of hundreds of thousands. These data originate from a heterogeneous collection of microservices within a cloud data center, which are in the form of spans that can be grouped into traces. A span is an immutable data structure that supplies the value of a collection of categorical and continuous variables at a particular point in time. The spans contain a *traceId*, based on which they can be grouped to form traces.

We consider each trace as a document where the span properties are considered as terms or words. The properties that are uninformative (meaning they are either same or unique for all the traces, *e.g., productName* or *id*) are ignored. The *traceId* is used only for trace reconstruction, not for trace representation. Continuous fields related to latency have not been used as they require non-standard distance measures for integration in an approach since the other variables are discrete. In addition, identifying latency anomalies falls into a separate class of problem, which is not in the scope of this research.

Since we are considering point anomalies [9], the trace document can be converted to a bag of words [8] which in essence is a count vector. We represent a trace as $x = (x_1, \ldots, x_d, \ldots, x_D)$, where D is the number of different terms that have been seen across all the traces in the dataset. For example, the HTTP code 200 is one term and a specific URL could be another one. Each dimension $x_d \geq 0$ is an integer value counting how many times a particular term appears in a trace. The resulting count data is agnostic of assumptions on the dataset, except knowledge of the total dimension D and what each dimension means. We can acquire such knowledge from an initial monitoring period. This knowledge base can also be updated in frequent intervals depending on the changes in microservices and their deployment architecture.

3.2 Comparing State-of-the-Art AD Methods

Since anomaly detection is a key step for a sampling process, we here illustrate which off-the-shelf anomaly detection methods could fit our purpose. We consider the following popular techniques: 1) local density estimate: *K-Nearest Neighbor* (KNN) and *Local Outlier Factor* (LOF), 2) tree-based classification: *Isolation Forest* and *Half Space Trees* (HST) [10], 3) boosting: *Lightweight Online Detection of Anomalies* (LODA) [20], and 4) neural network: *Deep Belief Net and One Class Support Vector Machine* (DBN+OCSVM) [21].

We have considered a production dataset from a cloud service operator consisting of 77577 traces from six consecutive days. The trace data have not been labelled by the operator. Thus, using the popular offline DBSCAN clustering algorithm, we have labeled the smallest clusters as anomalies, accounting for $\sim 5\%$ of the total traces. We evaluate the ability of the above-listed methods to obtain similar results.

Table 1 Results of different AD methods on the production dataset

	Isolation forest	KNN	LOF	LODA	DBN + OCSVM	HST
Precision	0.73	0.77	0.73	0.62	0.47	0.94
Recall	0.72	0.72	0.72	0.60	0.97	0.70
F1-score	**0.73**	**0.74**	**0.73**	**0.61**	**0.64**	**0.80**

The F1-Score, highlighted in bold, is highest for the HST method

The choice of DBSCAN is driven by the fact that it is considered as a generally reliable technique in industry [22].

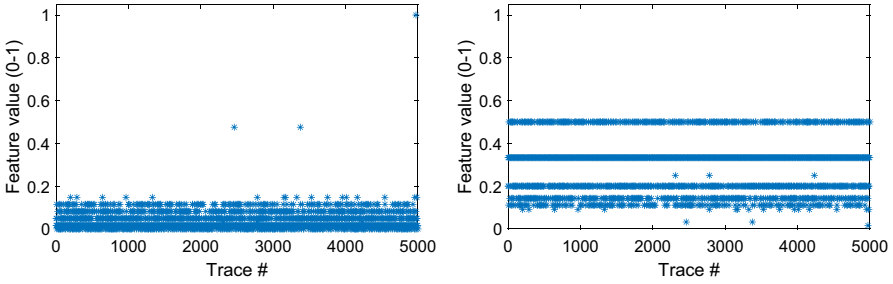
The results of the experiment are presented in Table 1. We see that HST is the best method with respect to F1-Score. This motivates further investigation into HST methods to address the problem under study. In addition, HST has other benefits from the perspective of a streaming platform. Due to the way HSTs are designed, for a particular trace, we only need to update a single mass value [23] per tree. To determine whether a trace is normal or anomalous, the mean mass value (m) of the HSTs, for that particular trace, is compared against a threshold. The use of HSTs also results in a very low computational footprint, as it only needs to query its already stored mass values.

3.3 Half Space Trees for Anomaly Detection

Half Space Trees (HST) [10] are an ensemble of binary decision trees, with depth d, where the corresponding binary tree has $2^{d+1} - 1$ nodes. Each tree stores split points among a random subset of dimensions, and possibly multiple splits per dimension. The leaves of the tree store how many points are within the subspace defined by the paths leading to them, which is referred as the *mass*. Since mass is defined as a count of data points, it is easier to calculate than density measures used in other methods, *e.g.*, those that require likelihood estimation. Normally, an ensemble of t Binary Trees is used, with identical depth h, which are independently trained on a data window w. In this study, we assume that the data points will be available for continuously arriving streams of spans from a heterogeneous collection of microservices.

In production, there could be relatively few types of traces that occur repeatedly. Thus, once the spans are vectorized to count data, this sparsity could cause the HST mass to be accumulated within a small set of terminal nodes. This is confirmed from our production data where only 0.004% of the trace count vectors are unique. We thus focus on a variant of HST known as HS*-Trees (HS*T) [23], which deals with the sparsity in the tree structure. In HS*T, nodes that have fewer than *SizeLimit* samples are not further expanded during the training phase. This reduces memory consumption and also the time to traverse the trees. Thus, we have opted for HS*T as our chosen HST variant. In the rest of the study, we use the terms HST and HS*T interchangeably.

Furthermore, we have incorporated two further modifications to HS*T. Firstly, we opted for depth-dependent split dimension. This means that when splitting a



(a) Min-Max Scaling (b) Transformation function f

Fig. 1 Comparing the scaled value of HTTP 200 code counts with min-max scaling and the transformation function f

node, instead of using the normal procedure of picking a dimension at random, we require all nodes at the same depth level to use the same split dimension, which largely reduces memory usage since a single dimension is stored at each level. Secondly, as suggested in [10], we opted for a [0, 1] workspace. This means that the maximum and minimum values of the features are assumed by the HS*T to be 1.0 and 0.0, rather than in the min-max range observed in the data. That is why for a better partitioning of the input data, as shown in Fig. 1, we have opted for the following transformation

$$f(x) = \frac{1}{1 + g(x)} \tag{1}$$

In our experiments, we have found it is sufficient to use $g(x) = x$, but other transformations could be used, *e.g.,* $g(x) = \log(x)$, to consider large values of x. We have used the production data to test these modifications, considering a day as a window and using the first day to build the trees. These changes improve the F1-Score from 0.8 to 0.97, indicating the changes aid in anomaly detection from the trace streams.

4 Mass-Based Clustering for Sampling

Although HSTs can help in classifying the anomalous traces, in reality, we need to utilize this classification output in a sampling process. This process is complex because of the trade-off between sampling normal and anomalous traces. While sampling, the proportion of the storage budget and the expected percentage of anomalies should be taken into account. If the budget is lower than the anomaly percentage, the focus should be on sampling mostly the anomalous traces. The normal traces should gain more attention only when the budget is higher than the anomaly percentage. In addition, while sampling the anomalous traces, the target should be representative sampling from that group of traces *i.e.* sampling from different "groups" of traces fairly.

To achieve this, we propose to cluster the traces and decide whether to sample a trace or not based on its cluster association. However, when clustering in a

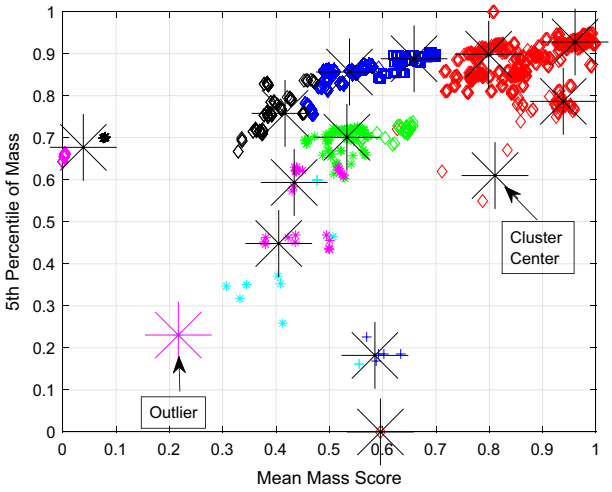


Fig. 2 The production trace plotted using the mass-based properties. The colors and markers indicate the DBSCAN clusters. The cluster centers are estimated with a baseline online clustering method

high-dimensional space it is harder to achieve accurate density estimation [19], in addition to incurring a higher computational cost. Therefore, we propose a new approach considering the distribution of mass across the trees in the HS*T forest and selecting a mean mass score m and a low percentile of the mass score p. Low percentiles are expected to significantly differ from the mean when there is at least a subset of trees in the forest that identifies the trace as an anomaly. We refer to this method as *SampleHST* as we are using the mass distribution of HST to perform sampling.

Since we want to use a low percentile (p) value along with the mean (m), we represent each trace with a unique pair (m, p) that will be used for clustering. The projection of the production traces in this 2-dimensional space is shown in Fig. 2. It shows the clusters, in different colors, obtained by DBSCAN. We see that the mass-based properties cluster the traces in distinct groups and the cluster centers are also appropriately detected using a baseline streaming clustering method [24]. Another potential benefit of using the low percentile value is a better separation of trace groups. As seen from Fig. 2, ignoring the percentile value will result in multiple trace groups being merged together, eventually affecting the sampling performance.

This mass-based clustering is at the core of SampleHST approach shown in Fig. 3. It starts the sampling process by receiving the spans related to a microservice. Its trace aggregator component composes them to form the traces and then convert them to either BoW-based or FGSD-based data format, depending on the choice of the SampleHST user. Finally, its sampler component decides whether to save the trace or not. Apart from the HSTs, the sampler contains a decision-making service, which is composed of two sub-components:

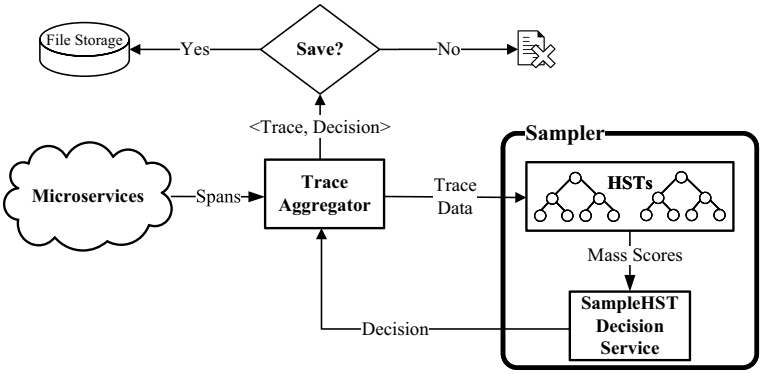


Fig. 3 An overview of the SampleHST approach

- **SampleHST Clustering**: Creates cluster of traces based on their mass based properties.
- **SampleHST Controller**: Makes the sampling decision based on budget and trace-cluster association.

We now discuss the details of these sub-components in the next sections.

5 SampleHST Clustering

SampleHST Clustering is primarily based on the underlying theory of *mean-shift analysis* [12] and the CEDAS algorithm [24], yielding a data-driven online approach that generalizes the hyper-sphere cluster shape commonly assumed in the literature to hyper-rectangles and hyper-cubes. Broadly speaking, our method receives the mass score of a trace in the form of a pair (m, p), which is generated using the HST mass distribution. Subsequently, the method aims to find the association of the new trace with an existing cluster, if the associated condition is not met a new cluster is created and a signal is sent. Furthermore, the method is able to remove clusters that have not received a new trace for a pre-defined period of time modulated by the *decay* and the *life (energy)* parameters. It can also merge clusters together whenever an overlapping occurs. These steps can be broadly grouped into two sets of tasks: *trace association* and *cluster management*. We now discuss the key aspects of these tasks.

5.1 Trace Association

Cluster Shape. A common assumption for online clustering algorithms for data streams is that the cluster shape is a hyper-sphere [12, 24]. However, in our setting, hyper-spheres can lead to inaccurate partitioning of the traces because the

normalized values of the unique pair (m, p) belong to the unit hyper-cube. To address this issue we consider instead an arithmetic average kernel whose support is a hyper-rectangle [25]. Assuming d data dimensions, the kernel considered is presented in (2) for which we further show in Theorem 1 that the mean-shift property is achieved if the clustering bandwidth [26] is equal in all dimensions.

Theorem 1 *For the additive kernel defined as*

$$K_d(u_1, \dots u_d) = \begin{cases} \frac{3}{d2^{d+1}} \sum_{k=1}^{d} \left(1 - u_k^2\right) & \text{if } |u_k| < 1, \forall k \\ 0 & \text{otherwise,} \end{cases} \tag{2}$$

the mean-shift algorithm at each iteration shifts each sample with a value equal to the local mean if the support is given by a hyper-cube.

Proof Assume that we have N observations $\{x_i\}_{i=1}^{N}$, with each data point having d dimensions, that is $x_i = (x_{i,1}, ..., x_{i,d})$, then we can define the density gradient estimate $\hat{\nabla} p(x) \equiv \nabla \hat{p}(x)$ at $x = (x_1, ..., x_d)$ as

$$\begin{aligned} \nabla \hat{p}(x) &= \frac{1}{Nh^d} \sum_{i=1}^{N} \nabla K\left(\frac{x_1 - x_{i,1}}{h}, \dots, \frac{x_d - x_{i,d}}{h}\right) \\ &= \frac{1}{Nh^d} \sum_{i=1}^{N} \sum_{k=1}^{d} \frac{\partial}{\partial x_k} K\left(\frac{x_1 - x_{i,1}}{h}, \dots, \frac{x_d - x_{i,d}}{h}\right) \cdot e_k \end{aligned} \tag{3}$$

where $\hat{p}(x)$ is the kernel density estimator [12, 25] for an unknown density p, h is the bandwidth in all d dimension, $K(\cdot)$ is the kernel function and e_k is the k-th standard unit vector. Now using (2) as our kernel function, we get

$$\begin{aligned} &\sum_{k=1}^{d} \frac{\partial}{\partial x_k} K\left(\frac{x_1 - x_{i,1}}{h}, \dots, \frac{x_d - x_{i,d}}{h}\right) \cdot e_k \\ &= \sum_{k=1}^{d} \frac{\partial}{\partial x_k} \left[\frac{3}{d2^{d+1}} \sum_{l=1}^{d} \left(1 - \left(\frac{x_l - x_{i,l}}{h}\right)^2\right)\right] \cdot e_k \\ &= \frac{3}{d2^{d+1}} \sum_{k=1}^{d} \frac{\partial}{\partial x_k} \left(1 - \left(\frac{x_k - x_{i,k}}{h}\right)^2\right) \cdot e_k \\ &= \frac{3}{d2^d h^2} \sum_{k=1}^{d} \left(x_{i,k} - x_k\right) \cdot e_k \end{aligned} \tag{4}$$

Substituting $\nabla K(\cdot)$ in (3) with (4), the density gradient estimate is

$$\begin{aligned}\nabla\hat{p}(x) &= \frac{1}{Nh^d}\frac{3}{d2^d h^2}\sum_{i=1}^{N}\sum_{k=1}^{d}\left(x_{i,k}-x_k\right)\cdot e_k \\ &= \frac{N_x}{Nh^d}\frac{3}{d2^d h^2}\sum_{k=1}^{d}\frac{1}{N_x}\sum_{x_{i,k}\in S_r(x)}\left(x_{i,k}-x_k\right)\cdot e_k \\ &= \frac{N_x}{Nh^d}\frac{3}{d2^d h^2}\sum_{k=1}^{d}\left(\mu_k-x_k\right)\cdot e_k\end{aligned} \tag{5}$$

Here, in (5), N_x is the number of data points for a region $A_h(x)$, μ_k is the mean of all the data points in that region along the k^{th} dimension. Note that, the volume of the region $A_h(x)$ is h^d and the probability density estimate $\hat{p}(x)$ over the region using a uniform kernel is $\frac{N_x}{Nh^d}$.

The objective of the mean shift algorithm is to move away from the valley and toward the function mode. This can be achieved through gradient ascent. That is, for two consecutive iteration t and $t+1$ the shift in the variable x_j in k^{th} dimension can be expressed as

$$x_{j,k}^{t+1} = x_{j,k}^{t} + c\frac{\nabla\hat{p}(x)}{\hat{p}(x)} \tag{6}$$

Now substituting the value of $\hat{p}(x)$ and $\nabla\hat{p}(x)$ and considering $c = \frac{d2^d h^2}{3}$ in (6), we show in (7) that the shift is equal to the mean of x in dimension k. This means that, from the online clustering perspective, with the arrival of each sample, the cluster center shifts u_k in dimension k.

$$x_{j,k}^{t+1} = x_{j,k}^{t} + \left(\mu_k - x_{j,k}^{t}\right) = \mu_k \tag{7}$$

Based on (7), we conclude that the mean shift algorithm is applicable to the kernel in (2) when the cluster bandwidth h is equal in all dimensions.

□

Cluster Assignation. The assignment step requires a pre-defined clustering bandwidth. We define the bandwidth vector, $H = \{h \in \mathbb{R}^d | \forall i = 1, \ldots, d,\ 0 < h_i \leq 1\}$, where each value $h_i \in H$ defines the Manhattan distance from the center to the boundary of the cluster in the i^{th} dimension. Now, if we define a vector of Manhattan distances between a cluster centroid and a new data point as $M = \{m \in \mathbb{R}^d | \forall i = 1, \ldots, d, 0 \leq m_i \leq 1\}$, then if $\forall i\ m_i \leq h_i$, we assign the data point to that cluster. Otherwise, a new cluster is created with that point.

Centroid Update. Appropriately updating the cluster centroid is critical since *SampleHST* uses the centroid distance to decide the mapping of traces to clusters. In general, it is preferable to update the centroid giving more importance to traces that are unequivocally within that cluster. This is the concept of *cluster kernel region* [24]. Given the clustering bandwidth vector $\mathbf{H}$, we can define the kernel region as the sub-space within a cluster with bandwidth $r\mathbf{H}$, where the scalar r quantifies the

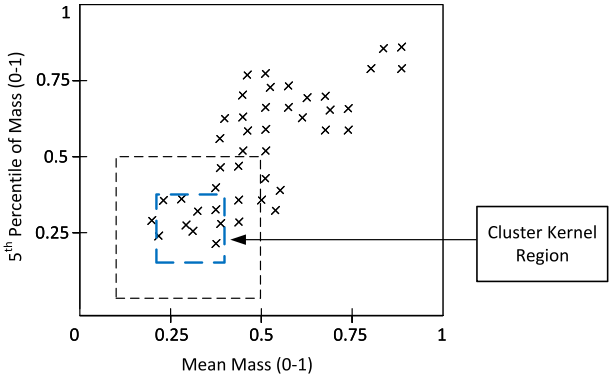


Fig. 4 Illustrating the kernel region of the cluster

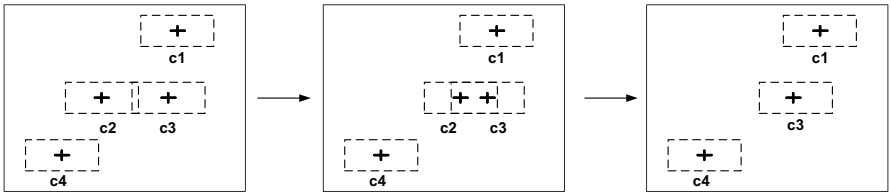


Fig. 5 Demonstrating cluster merging process. Initially, though there is an overlap between the boundary of cluster `c2` and `c3` they are not merged. Once their centroids overlap, they are merged into single cluster `c3`

proportion of the cluster considered as the kernel region. This concept of kernel region is illustrated in Fig. 4.

5.2 Cluster Management

Cluster Merging. To address the overlaps among clusters as they are indications of possibly inaccurate clustering, we opt for the policy that merges two clusters only when the centroid of one overlaps with the boundary of the other. This policy is less drastic than merging two clusters when their boundaries overlap because one distant point cannot shift the cluster center unless the cluster has very few samples. An illustration of this policy is presented in Fig. 5.

Cluster Removal. We need to regularly remove the clusters whose populations have remained static for a while since they are unlikely to be relevant and might affect the sampling policy. We realize this by using the *decay* and *life (energy)* parameters for the clusters as in [24]. The life property is initially set to one and gradually reduced using the decay value, which is set as the average number of traces in the work cycles, defined as a sequence of consecutive periods where we received at least 1 trace, within the sampling window. The overall cluster removal process is illustrated in Fig. 6.

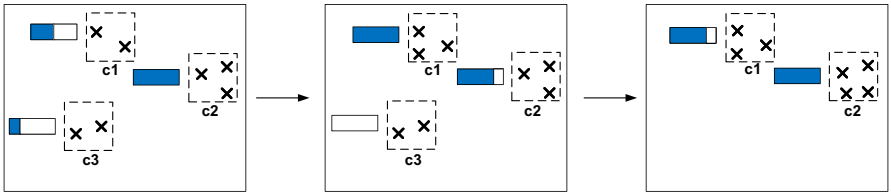


Fig. 6 Demonstrating the cluster removal process. There are 3 clusters initially and each of them has a life property. Each time there is a new observation, the life of the corresponding cluster is reset to full value, while the life of other clusters are reduced. A cluster is removed once its life ends. We can see that cluster `c3` is thus removed

Table 2 Auxiliary functions used in Algorithm 1

Function name	Description
SpawnCluster	Creates a new cluster with the current trace data.
PruneClusters	Removes the clusters whose life have expired.
FindAssociation	Finds the cluster where the current trace can be assigned. Returns -1 if no cluster is found.
AssignToCluster	Assigns the current trace to a cluster and update its center if necessary.
FindMergeCandidate	Finds whether the cluster associated with the current trace can be merged with any other cluster. Returns -1 if no cluster is found.
MergeClusters	Merges two clusters. The cluster centroid and its population is duly updated. Total number of clusters is reduced by 1.
DecayClusters	Reduces the life of each cluster by $\frac{1}{decay}$.

We present the overall algorithm for the clustering method in Algorithm 1. The functions that have been used in the algorithm is summarized in Table 2. This algorithm is utilized by the controller, which is presented in the next section.

Algorithm 1 Clustering

Input: trace (x), clusters (C), bandwidth (H), decay (γ)
Output: clusters(C), traceLocality (x_c)

if $C = \emptyset$ **then**
  $\{C, x_c\} \leftarrow \text{SpawnCluster}(C, x)$
else
  $C \leftarrow \text{PruneClusters}(C)$
  $initIdx \leftarrow \text{FindAssociation}(x, C, H)$
  if $initIdx$ = -1 **then**
    $\{C, x_x\} \leftarrow \text{SpawnCluster}(C, x)$
  else
    $C \leftarrow \text{AssignToCluster}(x, C, initIdx)$
    $mergeIdx \leftarrow \text{FindMergeCandidate}(C, H, initIdx)$
    if $mergeIdx > -1$ **then**
      $C \leftarrow \text{MergeClusters}(C, mergeIdx, initIdx)$
      if $mergeIdx > initIdx$ **then**
        $x_c = mergeIdx - 1$
      else
        $x_c = mergeIdx$
      end if
    else
      $x_c = initIdx$
    end if
  end if
  $C \leftarrow \text{DecayClusters}(C, \gamma)$
end if

6 SampleHST Controller

6.1 Overview

The SampleHST controller takes the sampling decision by utilizing the clustering method we have presented. The controller initially calculates the number of traces (s_w) that need to be sampled from the next sequence of w traces. We refer to this number as the sampling limit and the sequence as a window. For a given budget τ, the sampling limit is defined as $s_w = \tau w$. The budget is held constant, therefore the sampling limit only varies with w over the runtime. The sampling process runs continuously according to Algorithm 2, using HST mass scores x_m. The functions that we have used in the algorithm are summarized in Table 3. The algorithm expects a set of inputs that defines the size of the sampling window (w), the budget (τ), the total number of traces to be sampled in this window (s_w), the relative position of the current trace in the window ($w_i^{(p)}$), the number of traces that still remain to be sampled (s_r), the cluster status (C), which is a tuple

Table 3 Auxiliary functions used in Algorithm 2

Function name	Description
AdjustParameters	Adjusts the sampling limit if the current window size is larger than the estimation.
ScaleScores	Scales the mass scores using the min-max values. If the min-max value changes raises a flag.
HasMaxMinChanged	Check whether a flag is raised from the ScaleScores function. The details are provided in Sect. 6.2.
ReScaleClusterCenters	Re-scale the cluster centers using the previous and current min-max values. The details are provided in Sect. 6.2.
GetTraceLocality	Use Algorithm 1 to return the cluster index of current mass score.
IsTraceInSelectionPool	Determines whether the trace falls in any of the clusters in the *selection pool*.

containing the cluster centroids, cluster life values, and cluster sizes, the clustering bandwidth vector (H) and the length of the system work cycle (β).

Algorithm 2 Sampling

Input: massScores (x_m), budget (τ), idxPriWindow ($w_i^{(p)}$), windowSize (w), remainingTarget (s_r), windowTarget (s_w), clusters (C), bandwidth (H), workCycleLen (β)
Output: *decision*

if $w_i^{(p)} > w$ **then**
  AdjustParameters()
end if
$x_m^{(\log)} = \log_b(x_m)$
$x_m^{(s)} = \text{ScaleScores}(x_m^{(\log)})$
if HasMaxMinChanged() **then**
  ReScaleClusterCenters()
end if
$(C, x_c) \leftarrow \text{GetTraceLocality}(x_m^{(s)}, C, H, \frac{1}{\beta})$
$R = \frac{w_i^{(p)}}{w}$
$U = \frac{s_w - s_r}{s_w}$
if $s_r > 0$ **then**
  $decision = \text{IsTraceInSelectionPool}(C, x_c, \tau, R, U)$
end if
if *decision* **then**
  $s_r = s_r - 1$
end if

Initially, a series of pre-processing procedures takes place on the received data. Subsequently, the locality of the trace, represented by its associated cluster index, is determined by SampleHST clustering. The final step is the sampling decision based on the inclusion of the trace in a set of prioritized clusters, which we refer to as the

selection pool. This step is skipped if the sampling target has already been reached. Since we already discussed the SampleHST clustering method, we now present the key aspects of the controller.

6.2 Online Score Scaling

The first step in Algorithm 2 is to make the adjustments to the sampling window size estimate and the sampling target when the current window is larger than the expected window size. This is followed by log-transformation and min-max scaling of mass scores: $x_m^{(s)} = [\log_b(x_m) - \min(x_m^s)]/[\max(x_m) - \min(x_m)]$. It should be noted that before the log transformation, the mass scores are expected to be standardized. Since we are using HS*T, we use the mass value $m[l]2^l$, where $m[l]$ is the mass of the terminal node where the trace falls into and l is the depth of the corresponding tree node. To standardize the mass scores, we scale down the augmented mass using the maximum mass value possible, which is $w2^d$ where d is the tree depth and w is the number of observed traces.

Once the mass scores are processed, it is checked whether the minimum or maximum values change along with the new mass scores in the current sampling window. If this is the case, all the cluster centers are re-scaled.

6.3 Sampling Decision

The sampling decision procedure needs to decide on-the-fly whether to sample a trace or not. If a new cluster is created by a trace, then the method always samples it. For the case where the trace is associated with an existing cluster, we rely instead on generating a prioritized pool of clusters, which we refer to as *selection pool*, and use it to take the decision. This is done in three steps, which we describe in the following.

6.3.1 Distance-Based Cluster Ranking

The first step is to rank the clusters. Two methods of ranking are considered: size of the cluster and Euclidean distance from the origin. Cluster size is an obvious method of ranking, but since SampleHST creates and deletes clusters in an online manner, smaller clusters might not always represent less frequent traces. A cluster might be smaller but all of its traces can have high mass values. This means that the traces have hit HST nodes with a high mass count which indicates that these traces are quite frequent. In addition, the most interesting and possibly smallest clusters are likely to be near the origin, which represents a low-mass region in the clustering place. Therefore, we choose the Euclidean distance of the centroids to the origin (0, 0) and if a cluster is closer to the origin, traces associated with it will be sampled first even if that cluster is not the smallest. Thus, if a cluster is closer to the origin, as shown in Fig. 7, traces associated with it will be sampled first even if that cluster is not the smallest.

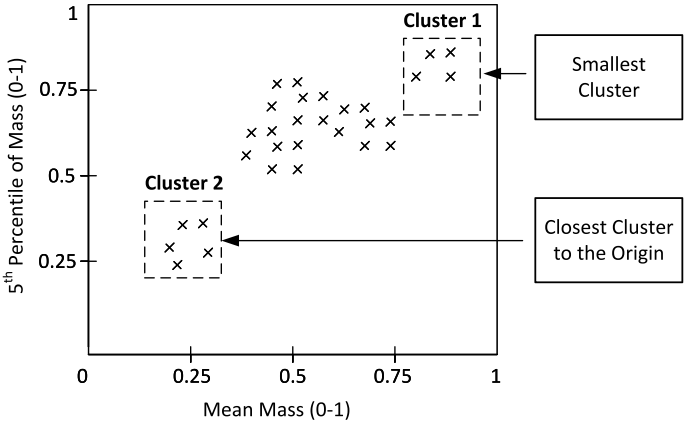


Fig. 7 Illustrating the distance based cluster ranking concept

6.3.2 Selection Pool

Once the clusters are ranked, we decide how many of those will form the initial selection pool. Clusters are added according to the above ranking, starting with the one closest to the origin, until the threshold θ is reached. If two clusters are equidistant, the one created first is prioritized.

After creating the initial selection pool, we start the second phase by checking the actual value of the percentage total population in the selection pool denoted by $\hat{\theta}$. If the actual percentage is less than $\alpha\%$ of the budget, we add more clusters to the selection pool. The clusters are added depending on the magnitude M of the budget (τ) in comparison to $\hat{\theta}$. This is defined as $M = \left\lfloor (\tau - \hat{\theta})/\hat{\theta} + \frac{1}{2} \right\rfloor$. We then make M independent attempts to add the clusters in a probabilistic manner, where in the k^{th} attempt, the k^{th} closest cluster to the origin, which is not yet included in the selection pool, is chosen with a probability P^k. Here each attempt of being successful has the same probability $P = \max(\tau, S)$, where τ is the budget and S is the sampling eagerness, bounded between [0, 1], defined as

$$S = R(1 - U) \tag{8}$$

A high eagerness value indicates we should sample more. It is defined in terms of the budget utilization (U), which is the ratio of the number of sampled traces to the sampling limit, and the relative trace position in the current window (R), which is the ratio of the trace index in the current window to the sampling window size.

6.3.3 Decision Process

After the selection pool has been decided, we sample the new trace only if it is associated with any of the clusters in the pool. If that is the case, one of two paths may be followed. If the budget is greater than or equal to the actual percentage of the

population in the selection pool ($\tau \geq \hat{\theta}$), we sample the trace straightaway. Conversely, if the budget is less than the actual percentage, we follow the second path that takes a probabilistic sampling decision. This is to sample cautiously as we may have larger clusters in the selection pool containing common traces. In this path, we set the probability of sampling as

$$P_s = \begin{cases} \frac{\tau}{\hat{\theta}} & \text{if } \Gamma_c > \Gamma_\mu + k\Gamma_\sigma \\ 1 & \text{otherwise} \end{cases} \tag{9}$$

Here we set the probability based on the cluster size. Firstly, if the size of the cluster (Γ_c), which is associated with the current trace, is greater than the sum of the mean (Γ_μ) and k standard deviation (Γ_{sigma}) of the cluster size in the selection pool, we set the sampling probability to $\tau/\hat{\theta}$. This means that, if there are N traces, the size of the selection pool will be $N\hat{\theta}$ and we would like to sample $N\tau$ traces from those in the selection pool. Secondly, if $\Gamma_c \leq \Gamma_\mu + k\Gamma_\sigma$, we set the sampling probability to 1. This means if the cluster is sufficiently small, we decide to sample the corresponding trace. The value of k is set using Chebyshev's inequality [27], which estimates the minimum percentage (V) of values within k standard deviation of the mean. For a given V, we can solve the inequality to determine the value of k. We notice that this percentage V is related to the ratio of $\tau/\hat{\theta}$. Because, if τ is much smaller than $\hat{\theta}$, we want to sample only if the associated cluster is smaller than the majority of the clusters. As the value of τ increases compared to $\hat{\theta}$, we can consider the larger clusters i.e., a larger value of V. Thus, considering $\hat{V} = \tau/\hat{\theta}$, where $\hat{V}$ is an estimate of the minimum percentage V, we can calculate the value of k using (10).

$$k = \sqrt{\frac{\hat{\theta}}{\hat{\theta} - \tau}} \tag{10}$$

7 Sampling Performance

7.1 Experimental Setup

To test the performance of SampleHST, we use a production data provided by a cloud data center composed of 77,577 traces. Each trace contains at least one span and the following four categorical features: *Service Name, URL, Process Id*, and *Node Id*. A span also contains the HTTP return code and HTTP method for the service invocation. The traces are represented as a count vector using the BoW model as detailed in Sect. 3. Through this, we obtain 105 unique features. To evaluate the performance of the trace samplers, we use the performance evaluation criteria mentioned in Sect. 2.2.

To test the SampleHST robustness, we consider 5 cases with different storage budgets. First, since we have about 5% anomalies in our data, we include a case where the budget is 5%. The evaluation criteria for this case is the F1-Score. We

have also chosen 3 smaller budgets (0.5%, 1% and 2%) where the evaluation criteria is precision. Finally, we also considered a high budget case of 10%, where the evaluation criteria is recall. We compared the results with two other samplers: uniform random sampler, implemented following the Head-based sampler in [5], and the PERCH-based method [3].

Since sampling methods such as [3, 5] focus on representative sampling, we also compare their fairness using the Jain index [15]. The index can be calculated using (11) where $X_i = \frac{T_i}{O_i}$. Here, for each cluster i, T_i is the number of traces sampled by a method and O_i is the optimal number of traces that should be sampled. This metric indicates what percentage of the groups are treated fairly. In our case, the groups are the clusters that we obtained offline from DBSCAN. We calculate the optimal number of traces that should be sampled offline using the max-min fair allocation approach [28].

$$\mathcal{J}(X_1, X_2, \ldots, X_n) = \frac{\left(\sum_{i=1}^{n} X_i\right)^2}{n \sum_{i=1}^{n} X_i^2} \quad X_i \geq 0 \tag{11}$$

7.2 Results

SampleHST Clustering Operation. We begin by illustrating in Fig. 8 the operation of the SampleHST method. Since this is an online clustering method, we divide the total time frame into 20 periods and show the clustering status for those periods. We immediately see that in the first window, the data points are less segregated. This is because of the online min-max scaling. In the initial period, the min-max values are not steady, which affects the data points as well. As we progress toward the end, we can see that the clusters are increasingly segregated. We also see that the number of clusters continues to change throughout these periods. The clusters around the top right corner remain stable, but the ones around the bottom left corner change their positions frequently as the top right clusters capture frequent traces whereas the bottom left ones capture infrequent traces. The infrequent trace clusters decay quickly by not receiving traces in some work cycles.

Comparative Experiments. We now compare the performance of SampleHST against the uniform random and PERCH-based methods. In Table 4 we see that SampleHST with a bandwidth of $h = 0.1$ is the best method across all budgets, with the uniform random sampler performing the worst. We also see that the PERCH-based method does not perform significantly better with respect the precision, recall, and F1-Score. From the fairness perspective, the PERCH-based method scores much higher than the random sampler, but still it cannot outperform SampleHST. The results show that even though the PERCH-based method can achieve a better

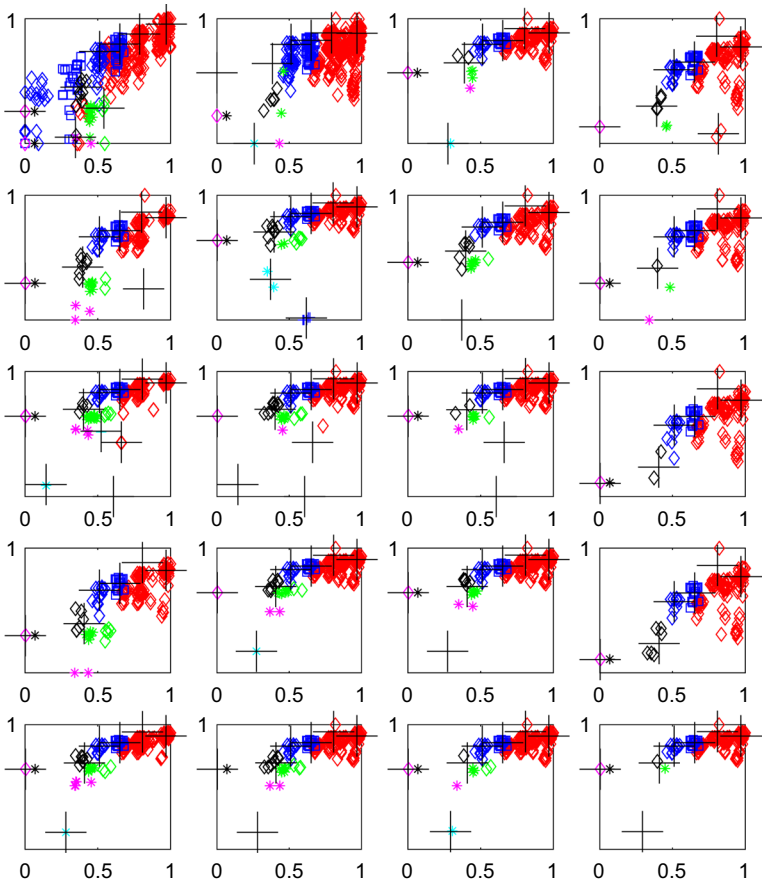


Fig. 8 Output of the SampleHST clustering algorithm. The X-axis and Y-axis represent mean and 5th percentile of mass respectively. The colored symbols represent different DBSCAN labels. The + signs are the cluster centers estimated by the SampleHST clustering algorithm. The output is presented in 20 windows. As we move from left to right, we move toward the next window

Jain score in low budgets, it is not precise in sampling the anomalous traces as made evident by the precision score.

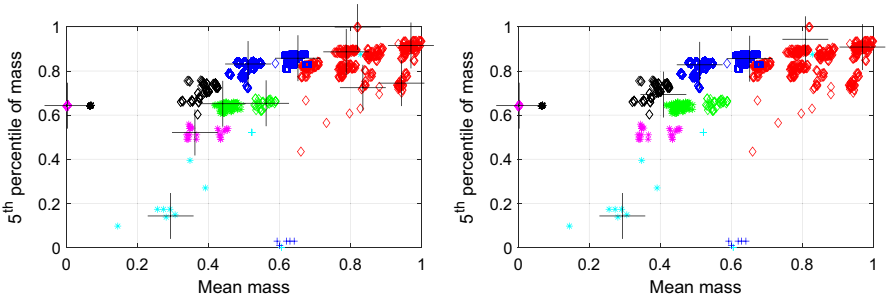
As we mentioned earlier, identifying anomalous traces is difficult for clustering methods due to the high number of dimensions of the input data, as in the present case with 105 dimensions. SampleHST, on the other hand, eliminates this problem by using the mass scores, which are low dimensional.

We now focus on the case with a high budget (10%). Firstly, we see that SampleHST easily outperforms the PERCH-based method considering the primary evaluation criteria recall. Secondly, when we consider representative sampling, we see that the Jain score produced by SampleHST is 1.6× better than the PERCH-based method. The reason for SampleHST performing better is as follows. The primary objective of SampleHST is to sample as many anomalous

Table 4 Performance of different samplers with different budget

		0.5%	1%	2%	5%	10%
Uniform	J	0.10	0.10	0.11	0.13	0.18
	P	**0.05**	**0.04**	**0.06**	0.05	0.05
	R	0.01	0.01	0.03	0.05	**0.10**
	F1	0.01	0.01	0.04	**0.05**	0.06
PERCH- based	J	0.32	0.24	0.32	0.47	0.56
	P	**0.41**	**0.18**	**0.13**	0.11	0.09
	R	0.03	0.03	0.04	0.09	**0.15**
	F1	0.05	0.04	0.07	**0.10**	0.11
SampleHST	J	0.40	0.59	0.72	0.75	0.88
	P	**0.84**	**0.83**	**0.86**	0.92	0.80
	R	0.10	0.18	0.37	0.91	**0.94**
	F1	0.17	0.30	0.52	**0.92**	0.87

The numbers corresponding to the evaluation criteria for different budgets are highlighted in bold



(a) $h_1 = h_2$ (b) $h_1 = h_2$

Fig. 9 Comparing clusters with equal and unequal clustering bandwidth

traces as possible. In high-budget cases, it only shifts focus towards normal traces when the primary objective is fulfilled. Anomalous traces can create many groups, each with a small size, whereas normal traces create a small number of large groups. This is indeed the case with the production data. As a result, when SampleHST samples most of the traces from anomalous groups, it satisfies the demands of the majority of the groups, making it fairer which is reflected in the Jain score.

SampleHST with Hyper-Rectangles. The mass scores work as anomaly signals to the SampleHST, which are not always likely to be equally strong in all clustering dimensions. In such cases, the traces may not be segregated ideally in that dimension. This is not a problem as long as we can separate anomalous traces from normal ones. However, if the bandwidth in that dimension is small, we can have multiple clusters in a particular region in the clustering hyper-plane, which represents traces of similar types. Thus rather than using a small clustering bandwidth in that dimension, as illustrated in Fig. 9, we can choose a large one to

Table 5 Performance of SampleHST considering hyper-rectangles

Bandwidth	Jain	Precision	Recall	F1-Score
0.05, 0.1	0.76	0.90	0.91	0.91
0.05, 0.2	0.75	0.91	0.91	0.91
0.05, 0.3	0.74	0.93	0.91	0.92
0.1, 0.2	0.74	0.94	0.91	0.92
0.1, 0.3	0.73	0.97	0.92	0.95

Bandwidths are highlighted in bold

Table 6 Sampling results with hyper-cubes and hyper-rectangles

	$h = 0.1$				$[h_1, h_2] = [0.1, 0.3]$			
	J	P	R	F1	J	P	R	F1
0.5%	0.40	**0.84**	0.10	0.17	0.41	**0.94**	0.10	0.18
1%	0.59	**0.83**	0.18	0.30	0.50	**0.95**	0.21	0.34
2%	0.72	**0.86**	0.37	0.52	0.47	**0.96**	0.41	0.58
5%	0.75	0.92	0.91	**0.92**	0.73	0.97	0.92	**0.95**
10%	0.88	0.80	**0.94**	0.87	0.88	0.79	**0.94**	0.86

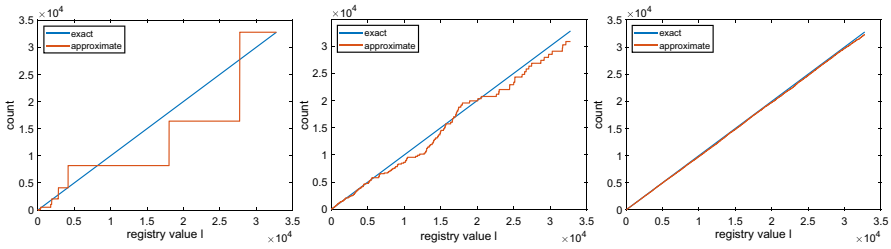
The budgets and numbers corresponding to the evaluation criteria for different budgets are highlighted in bold

remove clusters containing similar traces, allowing a more precise clustering. In other words, we can opt for hyper-rectangles instead of hyper-cubes.

When we observe the clustering status, as presented in Fig. 9, indeed with hyper-rectangles there are fewer clusters in the top right corner, that represent normal traces. Having less number of traces reduces the probability of sampling from normal groups, which is essential in the low and moderate budget cases. This is also reflected in the sampling performance. In Table 5 we present the results, for the 5% budget case and for different sizes of hyper-rectangles. From these results, we can appreciate that the F1-Score for bandwidth [0.1, 0.3] reaches 0.95, which is higher than the one we achieved for hyper-cubes presented in Table 4. Moreover, and considering the hyper-rectangle [0.1, 0.3] as our baseline we can see in Table 6 that the hyper-rectangles approach yields significantly better results in the metrics considered. In particular for low-budget scenarios we achieve on average an improvement of 1.12× with respect to hyper-cubes.

8 SampleHST-X: Integrating Approximate HST

As seen in Sect. 7, utilising HSTs clearly endows our sampling methodology with significant predictive performance. However, a possible drawback occurs when the HSTs grow very large stressing the memory capacity, so that there might be a need to save memory usage, even in the scale of tens of megabytes. In particular, if



(a) Morris' Algo., 4-bit (b) $\varepsilon = 0.10$, 9-bit (c) $\varepsilon = 0.01$, 14-bit

Fig. 10 Approximate counting methods: counting from 0 to 2^{15}

SampleHST-X is deployed to edge devices, which have memory constraints, reducing memory consumption in the scale of megabytes is also significant. Researchers have been working on better memory management of such devices either by proposing novel management method [29] or adopting the machine learning method [30]. We focus on the latter strategy and aim to reduce the memory footprint of HST. To address this issue, we consider *approximate counting* methods which are commonly used in high-speed network packet counting, see e.g. [31, 32].

The idea behind this novel approach, we propose, is to make the HS*Ts lighter by replacing the mass values stored in the HS*Ts nodes, with approximate counters that take less space since they require fewer bits to represent them. Theoretically, the key idea of approximate counting is to probabilistically increase the counter while ensuring that, in the long term, the bias is under control. This is usually achieved by introducing a probability indicating whether the counter should increase. In this direction, the traditional approach is the Morris' algorithm [13], where the probability to increase from l to $l+1$ is $1/D(l)$, with $D(l) = A(l+1) - A(l)$ and $A(l) = 2^l$. The function $A(l)$ is called the *estimation function*, which represents the value of the counter associated to the register value l. If the register ranges in $0, \ldots, L-1$, the maximum number that can be represented will be $A(L-1) = 2^{L-1}$. While Morris' algorithm allows for significant memory savings, it is fairly coarse-grained in the count values. Fig. 10a shows an example run of the Morris algorithm. By definition, the method takes as little as 4 bits to approximately represent numbers from 0 to 2^{15}. However, the approximation is increasingly worse the larger the numbers grow.

In recent years, new expressions for the $A(l)$ function have been proposed that trade accuracy for storage. In particular, the following estimation function is optimal [33]

$$A_\varepsilon(l) = \frac{(1+2\varepsilon^2)^l - 1}{2\varepsilon^2}(1+\varepsilon^2), \tag{12}$$

where ε is a parameter of the algorithm that needs to ensure that $A_\varepsilon(L-1)$ exceeds the desired maximum counter value M. In this case, the trade-off is characterised in terms of ε, so that for small values of ε, the approximation becomes better, whereas the storage saving will decrease. Examples for $\varepsilon = 0.10$ and $\varepsilon = 0.01$ are shown in Fig. 10b and 10c, where it can be appreciated the effect just described with $\varepsilon = 0.01$ giving the best approximation with a storage saving of just 1 bit.

Table 7 Anomaly detection results with approximate counting. All experiments use a HS*T with $\mathbf{T} = 500$ trees, depth $\mathbf{D} = 15$. We considered each day as a window and used the first day to built the trees

ε	bits	Precision	Recall	F1-score
Exact	17	0.9522	0.9853	0.9669
0.01	14	0.9522	0.9853	0.9669
0.10	9	0.9525	0.9848	0.9667
0.25	7	0.9458	0.985	0.9632
0.5	5	0.9164	0.9895	0.9482
0.75	4	0.8336	0.9952	0.9003

The different values of ε are highlighted in bold. Exact means approximate counting is not used

Table 8 Sampling results with exact and approximate mass ($h = 0.1$)

	Exact mass				Approximate mass			
	J	P	R	F1	J	P	R	F1
0.5%	0.404	0.840	0.093	0.167	0.424	0.829	0.091	0.164
1%	0.590	0.832	0.182	0.298	0.594	0.827	0.180	0.296
2%	0.723	0.860	0.372	0.520	0.726	0.860	0.372	0.520
5%	0.750	0.924	0.907	0.915	0.750	0.924	0.905	0.914
10%	0.88	0.807	0.943	0.869	0.879	0.808	0.941	0.869

The budgets considered are highlighted in bold

To test the approximate counting approach, we used the production data as before. The main objective of these experiments was to find the effect on the performance of the HST with and without approximate counting. This is particularly important, because this approach will make the mass values less precise and hence, having a negative impact on the F1 score of the anomaly detection method. The results of the experiments are shown in Table 7, where we can appreciate that for this data set the terminal node in a tree requires 17 bits to represent the mass value. Then, and as expected, with approximate counting, we appreciate that for larger values of ε, the F1-Score decreases. However, it is quite remarkable to see that we can still achieve an F1 score of 0.90 while saving 13 bits.

Of course, it is important to remark that this behaviour should not be expected in all the cases. For this particular data, we are able to see that the mass values are not uniform and can be easily segmented into a small number of groups. So that, high F1-Scores can be achieved if the high mass groups can be separated from the low mass groups. Overall, these results indicate that there is strong potential of compressing the space taken by HST with approximate counting, while not having such a dramatic negative effect on the F1-Score.

When we change the mass update mechanism to approximate from exact in SampleHST-X, we see no significant difference in performance, as shown in Table 8. This is because the scores produced by HST are qualitatively similar in both cases. Although approximate counting introduces errors in mass values, the normal and anomalous traces can still be separated. This is because of the definition of mass

Table 9 Memory usage of SampleHST-X using approximate and exact counting

Method	Size of mass value	Size of HST node	Memory usage	Runtime
Approximate	16 bits	24 bytes	79.3 MB	97 sec.
Exact	32 bits	30 bytes	94.4 MB	46 sec.

score $m[l]2^l$, where $m[l]$ is the mass of the corresponding node, which is in level l. Due to approximate counting, the error propagates through the term $m[l]$ but the 2^l part balances out the error, particularly in the case where the value of l is high.

Finally, to illustrate the impact of approximate counting, in reducing memory usage, we conduct another experiment. Here, we use a C++ programming language based implementation of SampleHST-X since it has a minimal abstraction from the underlying OS architecture and thus, has less memory overhead. We have profiled the memory usage (in terms of resident set size), for two versions of SampleHST-X, running on 30000 traces with 5000 trees. The first version with approximate counting used unsigned short integer, whose size is 16 bits, to store mass values. Here, we set the value of ϵ to 0.01. The second version with exact counting used unsigned integer, whose size is 32 bits, to store the mass values. The difference in memory usage is evident from Table 9. Based on this data, we see a 16% improvement in memory usage with approximate counting. It should be noted that the runtime for approximate counting is higher than exact counting, in-part due to the additional calculation involved with approximate counting. Considering the number of traces, the increase in runtime, in this case, is negligible. Nevertheless, this trade-off between memory saving and runtime should be considered when using approximate counting.

9 SampleHST-X: Extension for Collective Anomalies

In the previous sections, we focused on traces with point anomalies. However, a trace might not look anomalous from a point anomaly perspective as in such a perspective the collection of events are not considered [9]. To address this issue, we must consider that traces can contain collective anomalies. In particular, we focus on the sequence of events in a trace and how to consider the collection of those events during the anomaly detection process. We now present how we can incorporate this collective anomalies scenario using SampleHST-X. We demonstrate that, with only changing the trace representation, SampleHST-X produces a robust sampling performance for such scenarios.

9.1 Graph Model for Collective Anomalies

A single trace commonly spans across multiple microservices. The trace data can preserve this information about the call chain among the microservices. As a result, we can extract a workflow from the trace. How these call chains are preserved

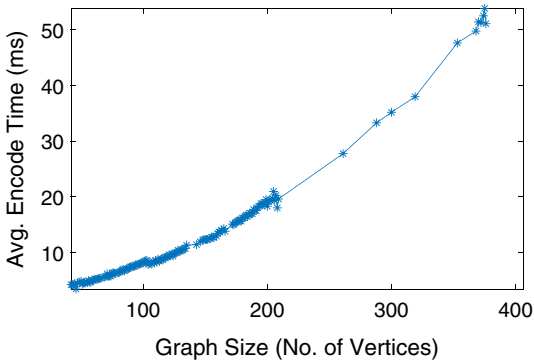


Fig. 11 Time complexity of FGSD algorithm with respect to graph sizes

depends on the trace format. For example, the Death Star Bench (DSB) traces follow the X-Trace format [34]. An X-Trace is composed of multiple events from different services, which makes is more fine-grained than the spans. Using the *source* property in each trace we can construct a directed acyclic graph (DAG) that represents the workflow among the services.

The Sifter method in [5] considers this DAG. It predicts the anomalies by inspecting all the k-length paths from the DAG. However, it ignores the overall graph, which in many cases is ideal for anomaly detection. For example, consider the traces related to the errors within a particular software component. These will mostly be structurally different from the typical traces. Thus, using an overall graph representation will possibly lead to better anomaly detection. Considering this, we need to use an unsupervised way to learn the features that represent the overall graph structure.

We have used the FGSD method [14] for this graph-based representation purpose. The main reasons, besides providing befitting graph features, for using FGSD is that it works in an unsupervised way and does not require to store previous traces to provide the current trace representation. FGSD is based on the spectral analysis of graphs. In such an analysis, the focus is to study the graph properties related to the set of graph eigenvalues. For FGSD the Laplacian matrix (the difference between the degree and adjacency matrix) is used to calculate the pairwise distances between the graph vertices. The final distance matrix is subsequently converted to a histogram, which is considered as the set of features. Since FGSD works on a simple connected graph, we first need to convert the DAG to its undirected form.

To implement FGSD in a real-time scenario we need to consider two issues, with the first one being scalability. Our main concern is the graph size and the time it requires to transform it to a feature vector. As mentioned already, the feature vector is a histogram of pairwise spectral distances. The number of histogram bins and range have no significant impact over the transformation time. However, it depends significantly on the number of vertices in the input graph. In the context of DSB dataset, it is the number of events. As reported in [14], the time complexity of FGSD is $\mathcal{O}(N^2)$ where N is the number of vertices in the graph, as such the method will not be applicable to all traces.

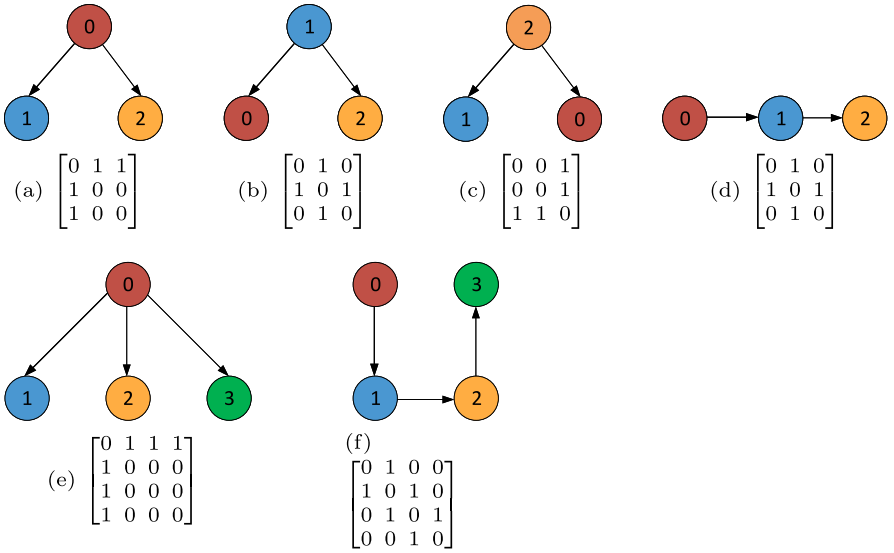


Fig. 12 Examples of directed graphs of traces and the adjacency matrix of their undirected version. The last two graphs correspond to traces of length four, the rest are of length three. Only the first three directed graphs are isomorphic, whereas the undirected versions of all the graphs are isomorphic

Although complexity of $\mathcal{O}(N^2)$ seems too onerous, we wanted to measure the maximum graph sizes that may still be processed in practical scenarios. We have tested this time complexity of FGSD using the DSB dataset. Here, we consider the traces whose corresponding graph size falls within 5^{th} to 95^{th} percentile. In other words, we ignored the graphs that are either too large or too small. We have tested each of those selected graphs 1000 times and considered the average time. We plotted the average time against the graph sizes in Fig. 11. We see that the initial growth in conversion time is roughly linear. The highest encoding time is around 50ms and till size 100, it is less than 10ms. Thus, adding this pre-processing layer in an anomaly detection process does not seem impractical if the graph sizes are such.

The second issue that we need to consider is graph isomorphism. The goal of FGSD is to determine whether two graphs have a similar structure. As a result, if two graphs are isomorphic, FGSD will provide a similar feature vector for them. Two graphs G and H are isomorphic when for every pair of vertices (u, v) that are adjacent in G, we can find a pair $(f(u),f(v))$ in H that are also adjacent where f is a bijection (one-to-one correspondence) between the vertices of G and H. In other words, *if we can relabel the nodes so that two graphs are identical, they are isomorphic*. This means that, despite having different parent-child relationship among the spans or traces in two separate traces, those two traces will have similar FGSD feature vector when the corresponding graphs are isomorphic. However, the method will be helpful if the traces differ in length and forking pattern.

For illustration purpose, let us consider three traces of length three (containing three spans). We present the corresponding directed graphs in Fig. 12a–c. The

figures also present the adjacency matrix of the corresponding undirected graphs, which is considered by FGSD. As seen in the figure, though the index of the root span is different, the structure of the graphs are similar and both the directed and undirected version of the graphs are isomorphic. Thus, FGSD will provide a similar feature vector for all these traces. In Fig. 12d we present a different orientation of the same spans. We can see that this directed graph is non-isomorphic to the directed graphs in Fig. 12a–c. However, the undirected version of the graph in Fig. 12d (as shown by the adjacency matrix), is isomorphic to the undirected version of the graphs in Fig. 12a–c. As a result, FGSD will yield the same feature vector for this trace as well.

Now we consider two more variants of the above mentioned traces where we add one extra span (shown in Fig. 12e-12f). As before, we also present the adjacency matrix of the corresponding undirected graphs in the figure. Here, it is clear that both the directed and undirected graphs of these traces are non-isomorphic. As a result their feature vector will be dissimilar and thus they can be differentiated by a anomaly detection algorithm. Note that, unlike FGSD, the count-based feature vector will yield the same feature vector for both traces.

9.2 Anomaly Detection

We now test how this FGSD-based trace representation aids the half space trees in anomaly detection. In this experiment we have used the DSB trace data. We particularly considered two APIs: *User Timeline* and *Compose Post*. For the User Timeline API, the median graph node size is 56. The Compose Post API, on the other hand, contains more complex traces with a median node size of 392. These two APIs give us a good mix of small, medium and large traces. They are also used for evaluation of the Sifter method [5], which we used for performance comparison. Here, similar to [5], we have mixed the normal and broken traces, and marked the broken traces as anomalies. Both datasets contain 1000 traces with 5% anomalies. Since there are more than 1000 traces available for both those APIs, we have randomly generated 30 combinations of them and report the average performance.

To understand the impact of FGSD on HST for anomaly detection, we consider the HST version for Sect. 3 *i.e.*, we do not apply the HST modifications (presented in Sect. 3) for this experiment only. As we want to compare the performance of HST with a method intended for collective anomalies, we use a variant of Sifter as an anomaly detection method. The concept behind this variant is as follows. Sifter provides a sampling probability for each trace, which is obtained from the loss while training a neural network. Thus, we can apply a threshold over this probability and when we observe a trace above this threshold, we mark it as an anomaly. We have selected this threshold on a trial and error basis to optimize the anomaly detection performance. Note that, the original Sifter method is not intended for anomaly detection but for sampling and it does not use a threshold. We discuss this in detail in the next subsection. We have implemented Sifter based on the loss function provided in

Table 10 Comparing the results of three methods on the DSB dataset

	User timeline			Compose post		
	P	R	F1	P	R	F1
Sifter	0.667	0.922	0.773	0.907	0.881	0.893
FGSD+DBN +OCSVM	0.919	0.943	0.931	0.845	0.889	0.857
FGSD+HST	0.961	0.999	0.980	0.949	0.944	0.946

Table 11 Sampling results for different budgets on the DSB dataset (compose post)

	HST			Approximate HST		
	P	R	F1	P	R	F1
0.5%	0.925	0.093	0.17	0.932	0.094	0.171
1%	0.948	0.191	0.317	0.949	0.191	0.318
2%	0.959	0.387	0.552	0.959	0.387	0.552
5%	0.965	0.914	0.939	0.965	0.913	0.938
10%	0.633	0.916	0.748	0.622	0.914	0.74

The budgets considered are highlighted in bold

the *word2vec* python library[4]. In addition to Sifter, we have incorporated a recent anomaly detection method DBN+OCSVM [21] with FGSD and also compared its result with others. The particular reason for choosing this method is that it provides the opportunity to compare the performance of another classifier other than HST, when combined with FGSD.

We present the comparison of these methods in Table 10. We see that for the Compose Post API, Sifter produces a slightly better F1-Score than FGSD + DBN + OCSVM. However, FGSD + DBN + OCSVM significantly outperforms Sifter for the User Timeline API. This shows that FGSD can provide a proper representation of a trace that can be used for anomaly detection. When we integrate FGSD with HST, we even achieve a higher F1-Score. FGSD + HST improves between 6% and 27% the F1-score of Sifter. HSTs are also 5% to 17% more accurate in F1-score than DBN + OCSVM.

Based on these results, it is clear that the FGSD-based trace representation and the HST classifier can detect the collective anomalies from trace streams. In the next section, we test whether this combination can produce accurate sampling performance.

9.3 Sampling Performance

We now apply the SampleHST-X method, with FGSD-based trace representation, on two datasets generated from the *Compose Post* and *User Timeline* part of the DSB traces. To assess the long-term sampling performance, each of those datasets

[4] Word2vec embeddings - https://radimrehurek.com/gensim/models/word2vec.html

Table 12 Sampling results for different budgets on the DSB dataset (user timeline)

	HST			Approximate HST		
	P	R	F1	P	R	F1
0.5%	0.892	0.092	0.166	0.891	0.091	0.166
1%	0.923	0.188	0.313	0.923	0.188	0.313
2%	0.949	0.386	0.549	0.949	0.386	0.549
5%	0.976	0.976	0.976	0.976	0.977	0.976
10%	0.976	0.999	0.987	0.976	0.999	0.987

The budgets considered are highlighted in bold

Table 13 Comparing the performance of different methods on the DSB dataset (Compose Post). The corresponding evaluation criteria is marked in bold

	Uniform			Sifter			SampleHST-X		
	P	R	F1	P	R	F1	P	R	F1
0.5%	**0.04**	0	0.01	**0.27**	0.04	0.07	**0.92**	0.09	0.17
1%	**0.05**	0.01	0.02	**0.29**	0.09	0.13	**0.95**	0.19	0.32
2%	**0.04**	0.02	0.02	**0.29**	0.17	0.21	**0.96**	0.39	0.55
5%	0.05	0.05	**0.05**	0.29	0.42	**0.35**	0.97	0.91	**0.94**
10%	0.04	**0.09**	0.06	0.27	**0.76**	0.40	0.63	**0.92**	0.75

Table 14 Comparing the performance of different methods on the DSB dataset (User Timeline). The corresponding evaluation criteria is marked in bold

	Uniform			Sifter			SampleHST-X		
	P	R	F1	P	R	F1	P	R	F1
0.5%	**0.05**	0	0.01	**0.06**	0.01	0.02	**0.89**	0.1	0.17
1%	**0.04**	0.01	0.01	**0.06**	0.02	0.03	**0.92**	0.19	0.31
2%	**0.05**	0.02	0.03	**0.05**	0.03	0.04	**0.95**	0.39	0.55
5%	**0.05**	**0.05**	**0.05**	0.05	0.08	**0.06**	0.98	0.98	**0.98**
10%	0.05	**0.1**	0.07	0.05	**0.15**	0.08	0.98	**1.0**	0.99

contains the same number of traces as the production data. We have injected 5% broken traces, which are marked as anomalies. At first, we present the results considering both HST and approximate HST. The results are presented in Table 11 and 12. Note that, we do not calculate the Jain score here as there are only two groups: normal and anomalies, which reduces the significance of fairness. From the results, we see that, in all the cases, SampleHST-X produces high values of corresponding evaluation criteria *i.e.*, the evaluation criteria mentioned in Sect. 2.2 in relation to budget. The values produced by approximate HST is roughly similar. Like the case for point anomalies, this similarity is due to the reason that the mass scores remain the same for both the exact and approximate HST.

We present the results comparing Sifter and SampleHST-X in Table 13 and 14. We see that for all the budgets, SampleHST-X outperforms the Sifter method. It may

Table 15 Sampling performance on the DSB traces (compose Post): sensitivity to cluster bandwidth

	Precision	Recall	F1-score
0.05,0.05	0.95	0.66	0.78
0.05,0.1	0.96	0.88	0.92
0.05,0.2	0.97	0.92	0.94
0.05,0.3	0.97	0.91	0.94
0.1,0.1	0.96	0.88	0.92
0.1,0.2	0.97	0.92	0.94

seem counter-intuitive to see that the sampling precision, recall, and F1-score for Sifter are much smaller than its anomaly detection values shown in Table 10, but it is not. The sampling values are smaller here because, as suggested for Sifter [5], we do not apply any threshold here and sample a trace using its sampling probability. However, since the sampling probabilities generated by Sifter are low, many of the anomalous traces are not sampled. This can be understood by analyzing the sampling probabilities generated by Sifter. For example, let us consider the experiments based on the User Timeline traces. We observe that the mean sampling probability generated by Sifter is 0.03. The 95^{th} percentile of the generated probability is 0.12, which is 4× higher than the mean. Sifter generally associated the anomalous traces with probability values above the 95^{th} percentile. Although the anomalous traces mostly had a sampling probability at least 4× higher than the normal ones, this still has not resulted in the anomalous traces being frequently sampled. A sampling probability of 0.12 means around 1 out of 8 anomalous traces will be sampled. Thus, for a small sampling budget, the sampling performance will be practically similar for Sifter and a uniform random sampler. This is evident from the results in Table 14.

We now present the sampling performance considering different clustering bandwidths with the Compose Post API. The results we have presented so far, in this section, is considered hyper-rectangles as this yielded the best results for point anomalies. For the Compose Post API, the used bandwidth is [0.1, 0.3] which yielded the best sampling performance for the production data in Sect. 7. In Table 15, we provide results considering both hyper-cubes and hyper-rectangles. Here, the average F1-Score is 0.91. This indicates that SampleHST-X is also robust to change in bandwidth when we consider the FGSD-based trace representation. We have one interesting observation here. The F1-Score for the smallest bandwidth [0.05, 0.05] is 14% lower than the average. The reason for this is with such small bandwidth, we can end up with many clusters. This can be a problem as we also have a probabilistic path in taking the sampling decision and some clusters may be ignored only because having a large number of clusters in the selection pool reduces their selection probability.

From these experiments, it is evident that, like point anomalies, using SampleHST-X, we can produce a robust sampling performance for collective anomalies. This is certainly encouraging as for SampleHST-X, the transition from point to collective anomalies can be done with minimal effort - we only need to change the trace model. Since both the trace models are based on count data, the rest of the SampleHST-X pipeline can remain the same.

10 Related Work

An important step for a sampler is to differentiate normal and anomalous traces *i.e.,* anomaly detection (AD). There are many recent works on AD for microservices using trace data. For example, the authors in [35–37] primarily learn from the patterns of call trees and request execution respectively to detect anomalies. Some studies [38–40] also consider deep learning based methods focusing on different aspects, *e.g.,* response times and causal relationships. However, these works do not consider transforming the anomaly detection result to a sampling decision.

To the best of our knowledge, there are only a few studies focusing on sampling anomalous traces generated by microservices. In [3], the authors proposed a sampler based on a hierarchical clustering method PERCH [17]. Although the method can potentially achieve representative sampling [3], it inherently incurs the curse of the data dimensionality during clustering [41] and requires batch processing, which is not preferred under low latency requirements.

Sifter [5] avoids batch processing by taking sampling decisions trace-by-trace. It relies on a sampling probability, generated by utilizing the loss of training a neural network for a particular trace. For such a loss-based method, anomalous traces may still have small probabilities overall, closer to 0 than to 1, allowing several anomalous traces to get not sampled. This problem is studied in a recently proposed sampler, Sieve [6], which uses a threshold to first separate the anomalous traces and then amplify the sampling probability. This still leaves an open challenge regarding the optimal and automated choice of threshold.

In our recent paper [7], we proposed the SampleHST approach that addresses the issues with probabilistic approaches and does not require batch processing. However, SampleHST itself can require large memory spaces, in the context of edge devices, to save the HST forest. In addition it does not consider collective anomalies [9]. We address these issues in this paper with the SampleHST-X method.

11 Threats to Validity

To the best of our knowledge, there is no dataset in trace sampling domain that includes both point and collective anomalies. Thus, we evaluated SampleHST-X with either point or collective anomalies in the traces. We did not generate a synthetic dataset for this purpose as developing such dataset, which represents a practical scenario, involves separate research challenges and it could be a possible future research direction. Once such data is available, the SampleHST-X pipeline requires no additional components to run the sampling process.

For the collective anomalies scenario, we compared the performance of SampleHST-X with Sifter. Recently another approach is proposed namely Sieve that unlike Sifter, depending on cases, can amplify the sampling probability. We do not implement Sieve to compare the results as, regardless of the amplification, the sampling process is still probabilistic making its approach similar to Sifter. In addition, the amplification requires determining a threshold which can significantly effect the

sampling performance. Thus, we only considered Sifter for the collective anomaly scenario.

12 Conclusion and Future Work

We have proposed *SampleHST-X*, a novel sampling method for distributed tracing with storage budget constraints. The goal of SampleHST-X is to incorporate the proportion of sampling budget and the fraction of expected anomalous traces while taking sampling decisions. For the case where the budget is lower, sampling the anomalous traces receives priority. On the other hand, when the budget is higher, the normal traces are sampled alongside the anomalous ones.

SampleHST-X relies on an online clustering mechanism using mass scores of the traces, which are generated using a forest of HST. Subsequently, if the budget allows, the sampling decisions are taken based on the association of a trace with a cluster, where the clusters more likely to contain anomalous traces are prioritized. Our experiments, which consider production data from a cloud service operator, show that our approach by far outperforms the recent approach targeting point anomalies. The incorporation of approximate HST yields similar results while reducing the space requirement of HST. A graph model, FGSD, based extension demonstrates SampleHST-X can produce high sampling accuracy considering collective anomalies as well.

A possible future work could be incorporating continuous trace properties, *e.g.*, the response time, to identify also the latency anomalies in an integrated approach.

Acknowledgements This research has received funding by Huawei Technologies (Ireland) Co., Ltd.

Author Contributions The author names and their contributions are listed below: A.G. Writing—Original Draft, Validation, Software, and Investigation; Y.G. Writing—Review and Editing, Validation, Software, and Investigation; M.S. Writing—Review and Editing, Validation, Software, and Investigation; J.P: Writing—Review and Editing, Validation, Software, and Investigation; O.O.: Conceptualization and Funding Acquisition; G.C.: Supervision, Writing—Review and Editing, Validation, Software, and Investigation.

Data Availability No datasets were generated or analysed during the current study.

Declarations

Competing interest The authors declare no competing interests.

References

1. Richardson, C.: Microservices Patterns: With Examples in Java. Simon and Schuster, London (2018)
2. Guo, X., Peng, X., Wang, H., Li, W., Jiang, H., Ding, D., Xie, T., Su, L.: Graph-based trace analysis for microservice architecture understanding and problem diagnosis. In: Proceedings of ESEC/FSE, pp. 1387–1397 (2020). ACM
3. Las-Casas, P., Mace, J., Guedes, D., Fonseca, R.: Weighted sampling of execution traces: capturing more needles and less hay. In: Proceedings of SoCC, pp. 326–332. ACM (2018)
4. Parker, A., Spoonhower, D., Mace, J., Sigelman, B., Isaacs, R.: Distributed Tracing in Practice: Instrumenting, Analyzing, and Debugging Microservices. O'Reilly Media, Sebastopol (2020)
5. Las-Casas, P., Papakerashvili, G., Anand, V., Mace, J.: Sifter: scalable sampling for distributed traces, without feature engineering. In: Proceedings of SoCC, pp. 312–324. ACM (2019)
6. Huang, Z., Chen, P., Yu, G., Chen, H., Zheng, Z.: Sieve: Attention-based sampling of end-to-end trace data in distributed microservice systems. In: Proceedings of ICWS, pp. 436–446. IEEE (2021)
7. Gias, A.U., Gao, Y., Sheldon, M., Perusquía, J.A., O'Brien, O., Casale, G.: Samplehst: efficient on-the-fly selection of distributed traces. In: NOMS 2023-2023 IEEE/IFIP Network Operations and Management Symposium, pp. 1–9. IEEE (2023)
8. Zhang, Y., Jin, R., Zhou, Z.-H.: Understanding bag-of-words model: a statistical framework. Int. J. Mach. Learn. Cybern. **1**(1-4), 43–52 (2010)
9. Chandola, V., Banerjee, A., Kumar, V.: Anomaly detection: a survey. ACM Comput. Surv. **41**(3), 15–11558 (2009)
10. Tan, S.C., Ting, K.M., Liu, T.F.: Fast anomaly detection for streaming data. In: Proceedings of IJCAI, pp. 1511–1516 (2011)
11. Fukunaga, K., Hostetler, L.: The estimation of the gradient of a density function, with applications in pattern recognition. IEEE Trans. Inf. Theory **21**(1), 32–40 (1975)
12. Baruah, R.D., Angelov, P.: Evolving local means method for clustering of streaming data. In: Proceedings of FUZZ-IEEE, pp. 1–8. IEEE (2012)
13. Morris, R.: Counting large numbers of events in small registers. Commun. ACM **21**(10), 840–842 (1978)
14. Verma, S., Zhang, Z.-L.: Hunt for the unique, stable, sparse and fast feature learning on graphs. In: Proceedings of NIPS, pp. 88–98 (2017)
15. Jain, R.K., Chiu, D.-M.W., Hawe, W.R.: A Quantitative Measure of Fairness and Discrimination for Resource Allocation in Shared Computer System. Eastern Research Laboratory, Digital Equipment Corporation, Hudson, MA (1984)
16. Gan, Y., Zhang, Y., Cheng, D., Shetty, A., Rathi, P., Katarki, N., Bruno, A., Hu, J., Ritchken, B., Jackson, B., et al.: An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In: Proceedings of ASPLOS, pp. 3–18 (2019)
17. Kobren, A., Monath, N., Krishnamurthy, A., McCallum, A.: A hierarchical algorithm for extreme clustering. In: Proceedings of SIGKDD, pp. 255–264 (2017)
18. Le, Q., Mikolov, T.: Distributed representations of sentences and documents. In: Proceedings of ICML, pp. 1188–1196. PMLR (2014)
19. Fukunaga, K.: Introduction to Statistical Pattern Recognition. Elsevier, Amsterdam (2013)
20. Pevnỳ, T.: Loda: lightweight on-line detector of anomalies. Mach. Learn. **102**(2), 275–304 (2016)
21. Erfani, S.M., Rajasegarar, S., Karunasekera, S., Leckie, C.: High-dimensional and large-scale anomaly detection using a linear one-class SVM with deep learning. Pattern Recogn. **58**, 121–134 (2016)
22. Fisher-Ogden, P., Burrell, G., Sanden, C., Rioux, C.: Tracking down the Villains: outlier detection at Netflix. https://netflixtechblog.com/tracking-down-the-villains-outlier-detection-at-netflix-40360b31732. Accessed 25 Jan 2023 (2015)
23. Ting, K.M., Zhou, G.-T., Liu, F.T., Tan, S.C.: Mass estimation. Mach. Learn. **90**(1), 127–160 (2013)
24. Hyde, R., Angelov, P., MacKenzie, A.R.: Fully online clustering of evolving data streams into arbitrarily shaped clusters. Inf. Sci. **382**, 96–114 (2017)
25. Langrené, N., Warin, X.: Fast and stable multivariate kernel density estimation by fast sum updating. J. Comput. Graph. Stat. **28**(3), 596–608 (2019)
26. Wand, M.P., Jones, M.C.: Kernel Smoothing. CRC Press, Boca Raton, Florida (1994)
27. Feller, W.: An Introduction to Probability Theory and Its Applications, vol. 2. John Wiley & Sons, Toronto (2008)

28. Jaffe, J.: Bottleneck flow control. IEEE Trans. Commun. **29**(7), 954–962 (1981)
29. Al-Maitah, M., AlZubi, A.A., Alarifi, A.: An optimal storage utilization technique for IoT devices using sequential machine learning. Comput. Netw. **152**, 98–105 (2019)
30. Lin, J., Zhu, L., Chen, W.-M., Wang, W.-C., Gan, C., Han, S.: On-device training under 256kb memory. Adv. Neural Inf. Process. Syst. **35**, 22941–22954 (2022)
31. Ikada, S., Hamaguchi, Y.: Approximate frequency counts algorithm for network monitoring and analysis: Improvement of "lossy counting". In: 2009 First International Conference on Emerging Network Intelligence, pp. 9–14 (2009)
32. Cvetkovski, A.: An algorithm for approximate counting using limited memory resources. SIGMETRICS Perform. Eval. Rev. **35**(1), 181–190 (2007)
33. Einziger, G., Fellman, B., Kassner, Y.: Independent counter estimation buckets. In: Proceedings of INFOCOM, pp. 2560–2568. IEEE (2015)
34. Fonseca, R., Porter, G., Katz, R.H., Shenker, S.: X-trace: A pervasive network tracing framework. In: Proceedings of NSDI, pp. 271–284 (2007)
35. Wang, T., Zhang, W., Xu, J., Gu, Z.: Workflow-aware automatic fault diagnosis for microservice-based applications with statistics. IEEE Trans. Netw. Serv. Manag. **17**(4), 2350–2363 (2020)
36. Zuo, Y., Wu, Y., Min, G., Huang, C., Pei, K.: An intelligent anomaly detection scheme for micro-services architectures with temporal and spatial data analysis. IEEE Trans. Cogn. Commun. Netw. **6**(2), 548–561 (2020)
37. Meng, L., Ji, F., Sun, Y., Wang, T.: Detecting anomalies in microservices with execution trace comparison. Future Gener. Comput. Syst. **116**, 291–301 (2021)
38. Nedelkoski, S., Cardoso, J., Kao, O.: Anomaly detection and classification using distributed tracing and deep learning. In: Proceedings of CCGRID, pp. 241–250. IEEE (2019)
39. Nedelkoski, S., Cardoso, J., Kao, O.: Anomaly detection from system tracing data using multimodal deep learning. In: Proceedings of CLOUD, pp. 179–186. IEEE (2019)
40. Bogatinovski, J., Nedelkoski, S., Cardoso, J., Kao, O.: Self-supervised anomaly detection from distributed traces. In: Proceedings of UCC, pp. 342–347. IEEE (2020)
41. Zimek, A., Schubert, E., Kriegel, H.-P.: A survey on unsupervised outlier detection in high-dimensional numerical data. Stat. Anal. Data Min.: ASA Data Sci. J. **5**(5), 363–387 (2012)

Alim Ul Gias is a lecturer at the Department of Computer Science in City, University of London. His current research focuses on QoS and resource management of cloud native applications, e.g., microservices, with queueing models and machine learning. He started working on this domain when he started his PhD in Imperial College London and continued his work while he was a post-doctoral research associate at the Centre for Parallel Computing, University of Westminster. Alim also has a background in software engineering, which helps him in designing solutions to problems with a complex system architecture.

Yicheng Gao received the B.Sc. degree in communication engineering from the University of Science and Technology Beijing, 2017, and the M.Sc. degree in information and communication engineering from Beijing University of Posts and Telecommunications, 2020. She is currently pursuing the PhD degree in computing with Imperial College London. Her research interests mainly include edge computing, caching, queueing modeling, and resource management.

Matthew Sheldon received the BS degree in Industrial and Systems Engineering from Georgia Tech in 2019 and the MSc degree in Computing at Imperial College London in 2021. He is now a PhD student in Computing at the same institution. His research interests include queueing models, mobility systems, pricing, and reinforcement learning.

José A. Perusquía is a postdoctoral researcher at the Department of Probability and Statistics, Institute for Research in Applied Mathematics and Systems, UNAM, Mexico City. He received his PhD degree in Statistics from the University of Kent in 2022. Before that he obtained his BS in Actuarial Science and MSc degree in Mathematical Sciences at UNAM. His research interest includes Bayesian statistics and

applied probability with emphasis on anomaly detection models in cyber security and computer systems.

Owen O'Brien is the chief cloud architect at Huawei Technologies, Dublin, Ireland. He currently leads the team for R&D for Autonomous Cloud Native Platforms to support Autonomous Driven Networks. He led several breakthrough projects that includes intelligent solutions such as fault detection/prediction, intelligent tracing, root-cause analysis, self-healing, etc. Prior to joining Huawei, Owen was a senior software development manager for the Microsoft's Azure platform. His team focused on developing features that drives reliability and availability for the Azure Traffic Manager.

Giuliano Casale Giuliano Casale joined the Department of Computing at Imperial College London in 2010, where he is currently a Reader. He does research in performance engineering and cloud computing, topics on which he has published more than 150 refereed papers. He has served on the technical program committee of several conferences in the area of performance and dependability. His research work has received multiple recognitions, including best paper awards at ACM SIGMETRICS, IEEE/IFIP DSN, and IEEE INFOCOM. During 2019-2023 he was the chair of ACM SIGMETRICS. He serves on the editorial boards of ACM TOMPECS and as Editor-in-Chief of Elsevier Performance Evaluation.

Authors and Affiliations

Alim Ul Gias[1] · Yicheng Gao[2] · Matthew Sheldon[2] · José A. Perusquía[3] · Owen O'Brien[4] · Giuliano Casale[2]

✉ Alim Ul Gias
alim.gias@city.ac.uk

Yicheng Gao
y.gao20@imperial.ac.uk

Matthew Sheldon
matthew.sheldon20@imperial.ac.uk

José A. Perusquía
jose.perusquia@sigma.iimas.unam.mx

Owen O'Brien
owen.obrien@huawei.com

Giuliano Casale
g.casale@imperial.ac.uk

[1] City, University of London, London, UK

[2] Imperial College London, London, UK

[3] Universidad Nacional Autónoma de México, Mexico City, Mexico

[4] Huawei Technologies (Ireland) Co. Ltd., Dublin, Ireland