The Journal of Systems & Software 202 (2023) 111708

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Software runtime monitoring with adaptive sampling rate to collect representative samples of execution traces[☆]

Jhonny Mertz[*], Ingrid Nunes

Universidade Federal do Rio Grande do Sul (UFRGS), Instituto de Informática, Porto Alegre, Brazil

ARTICLE INFO

Article history:
Received 23 July 2022
Received in revised form 25 February 2023
Accepted 12 April 2023
Available online 19 April 2023

Keywords:
Monitoring
Logging
Execution trace
Sampling
Adaptation
Self-adaptation

ABSTRACT

Monitoring software systems at runtime is key for understanding workloads, debugging, and self-adaptation. It typically involves collecting and storing observable software data, which can be analyzed online or offline. Despite the usefulness of collecting system data, it may significantly impact the system execution by delaying response times and competing with system resources. The typical approach to cope with this is to filter portions of the system to be monitored and to sample data. Although these approaches are a step towards achieving a desired trade-off between the amount of collected information and the impact on the system performance, they focus on collecting data of a particular type or may capture a sample that does not correspond to the actual system behavior. In response, we propose an adaptive runtime monitoring process to dynamically adapt the sampling rate while monitoring software systems. It includes algorithms with statistical foundations to improve the representativeness of collected samples without compromising the system performance. Our evaluation targets five applications of a widely used benchmark. It shows that the error (RMSE) of the samples collected with our approach is 9%–54% lower than the main alternative strategy (sampling rate inversely proportional to the throughput), with 1%–6% higher performance impact.

1. Introduction

Software runtime monitoring (Gao et al., 2017) is fundamental for ensuring software quality (Kang, 2018). It collects and often stores observable software data to enable, e.g., runtime verification, debugging, program comprehension, and self-adaptation (Feng et al., 2018). A typical type of software runtime data consists of execution traces (Pirzadeh et al., 2011; Yuan et al., 2014; Reger and Havelund, 2016). An *execution trace* is a record that provides information about the execution of software systems. These traces capture system operations, such as method calls or message exchanges, and include information of interest (e.g., method inputs and outputs) depending on the monitoring goal. Such goals can be the validation of quality requirements (Finocchi, 2013), identification of security vulnerabilities (Yuan et al., 2014) or model inconsistencies (Bartocci et al., 2018), performance engineering (Toffola et al., 2015; Mertz and Nunes, 2018), and optimization (Feng et al., 2018). AWS X-ray,[1] for example, is a tool that uses tracing execution to provide an end-to-end view of request paths in software applications, including a map of the application's underlying components. When an exception occurs while the application is serving an instrumented request, AWS X-ray records details about the exception, including the stack trace. This helps, e.g., identify and troubleshoot the root cause of performance issues and errors.

Despite the usefulness of execution traces, collecting them at runtime consumes resources and may cause performance decays (Bartocci et al., 2018), mainly when they include detailed information, such as method parameters. To address this, execution traces can be sampled or filtered. *Filtering* and *sampling* execution traces have been commonly adopted with pre-defined and fixed configurations, which specify certain software locations to be monitored and/or a sampling rate (Hamou-Lhadj and Lethbridge, 2004; Pirzadeh et al., 2011, 2013; Las-Casas et al., 2018). These configurations may be unsuitable to cope with software usage peaks and unable to handle unforeseen scenarios. AWS X-ray, for instance, applies a conservative sampling strategy and records only the first request of each second and five percent of any additional requests. Any different strategy must be manually managed, considering the performance impact it may cause to the application. These limitations are addressed by *adaptive* approaches (Zavala et al., 2019). However, existing work either focuses only on collecting traces for a particular purpose (Fei and Midkiff, 2006; Las-Casas et al., 2018) or uses a strategy that

[☆] Editor: Burak Turhan.

[*] Correspondence to: Instituto de Informática, Universidade Federal do Rio Grande do Sul (UFRGS), Av. Bento Gonçalves 9500, Bloco IV, Postal Code: 91501-970, Porto Alegre, Brazil.
E-mail addresses: jmamertz@inf.ufrgs.br (J. Mertz), ingridnunes@inf.ufrgs.br (I. Nunes).

[1] https://aws.amazon.com/xray/

https://doi.org/10.1016/j.jss.2023.111708
0164-1212/© 2023 Elsevier Inc. All rights reserved.

cannot guarantee that the collected traces are a *representative* sample of the population (Hauswirth and Chilimbi, 2004; Brönink and Rosenblum, 2016). This can potentially cause wrong decision making based on the sample or missing information.

To address the challenge of obtaining a representative sample while monitoring software systems, we propose an adaptive sampling process to collect execution traces with detailed information in environments where the performance impact is critical, such as production environments. Our goal is to pursue the representativeness of the samples of execution traces while adjusting the sampling rate used to monitor a software system to cope with increases in its workload. The process decides whether the operations executed to respond to each incoming request should be recorded as execution traces. Our process is composed of three activities: (1) *sampling decision*, which decides whether a request (with associated execution traces) should be recorded and included in a sample; (2) *sampling rate adaptation*, which adjusts the sampling rate at runtime; and (3) *sample evaluation*, which assesses the representativeness of the sample to identify the end of the monitoring cycle. These activities include algorithms with statistical foundations to ensure that, at the end of each monitoring cycle, the collected sample is representative of the population. Our evaluation targets five applications of the well-known DaCapo benchmark, which includes workloads that are used to create usage variations that occur in production environments such as stationary usage and spikes. The root-mean-square error (RMSE) of the samples collected with our approach is 9%–54% lower than the main alternative strategy (sampling rate inversely proportional to the throughput), with 1%–6% higher performance impact.

Because the intervention made in our evaluation consists solely in varying sampling rates, our conclusion provides reliable evidence that our adaptive sampling process is effective and efficient to be used while monitoring software systems. Our proposed process can thus be integrated as part of monitoring solutions, which can also include other mechanisms, such as filtering. The choice for suitable mechanisms for monitoring a particular software system depends, however, on the monitoring goal, e.g. runtime adaptation or anomaly detection.

The remainder of the paper is organized as follows. Next, we discuss existing software monitoring approaches. Our proposed solution is introduced in Section 3 and its empirical evaluation is presented in Section 4. Finally, we conclude in Section 5.

2. Related work

Monitoring executions traces with detailed information requires code instrumentation (Cassar et al., 2017), has a performance impact, and often consumes extensive storage space (Mertz and Nunes, 2017). Filtering and sampling have been demonstrated as practical mechanisms to reduce the monitoring overhead and enable faster trace analysis (Bartocci et al., 2018). While *filtering* excludes from the monitoring particular executions or software locations that are not of interest for a given goal (e.g. troubleshoot performance issues or find bugs), *sampling* involves establishing a sampling rate to monitor a subset of execution traces assuming that it is a representative sample of the population of traces. These two approaches can be combined to keep the monitoring overhead at acceptable levels. There is existing work to support the specification of filtering and sampling configurations, which give the scope of monitoring (i.e. locations to monitor) and the sampling rate, respectively. Proposed approaches can be classified into two groups: (i) *fixed configuration*, which keeps the same configuration throughout the software execution until it is manually updated; (ii) *adaptive configuration*, which dynamically adjusts the configuration based on constraints and lightweight monitored data. These groups are discussed as follows.

2.1. Fixed configuration

A straightforward way to cope with the monitoring overhead is to use random or systematic sampling (Chan et al., 2003; Dugerdil, 2007; Jung et al., 2014; Zhou et al., 2016; Song and Lu, 2017), which is used in various commercial and open-source tools (van Hoorn et al., 2012; Horký et al., 2016). However, as there are traces that are not recorded, important traces may be missed. Thus, choosing a sampling rate is a challenge (Las-Casas et al., 2018; Miranskyy et al., 2016; Mertz and Nunes, 2019, 2021) because of the trade-off between the representativeness of the sample and the performance overhead. A suitable solution in some scenarios is the use of a fixed (higher) sampling rate but targeting particular executions or regions of an application. Nevertheless, when the population of traces is not homogeneous, focusing on statically defined regions may lead to reduced coverage and thus an unrepresentative sample (Pirzadeh et al., 2013).

To support specifying a filtering configuration, there are approaches that perform an automated offline analysis of the program to define relevant application regions or paths (Apiwattanapong and Harrold, 2002; Santelices et al., 2006; Sridharan et al., 2007; Narayanappa et al., 2010). Although helpful, these solutions are not suitable when the areas of interest vary at runtime. In these cases, the approaches should be re-executed to tune the monitoring configuration.

2.2. Adaptive configuration

Adaptive monitoring approaches change, at runtime, the sampling and filtering configuration and even collected metrics. Work in this direction has been recently investigated in a systematic mapping (Zavala et al., 2019) that reveals that most of the proposed adaptations focus on improving the monitoring results for a *specific purpose*. Fei and Midkiff (2006), for example, observed that executing a (region of an) application with the same context tends to produce the same outcome. Therefore, repeated executions do not need to be monitored when the goal is to identify where bugs are likely to occur. Similarly, Las-Casas et al. (2018, 2019) aim at capturing outliers and anomalous traces. They aim to maximize the diversity of execution traces in the sample with infrequent patterns by computing the distance among traces and ensuring diversity in the set of traces, given a fixed budget. Targeting performance, Ding et al. (2015) proposed a cost-aware logging mechanism that decides whether to keep log messages based on (1) a dynamic measurement of the performance of the code snippet that generated the log and (2) an allowed maximum volume of logs in a time interval. The goal is to keep logs related to code snippets that execute slower than in the past. In a different direction, *Tigris* (Mertz and Nunes, 2019, 2021), a two-phase monitoring framework, can be instantiated considering a specific monitoring goal. The framework includes a domain-specific language to specify the criteria used in the first phase of the framework to dynamically choose the application locations that should be monitored in detail in the second phase.

These aforementioned adaptive approaches focus on dynamic filtering and adopt a fixed sampling rate—some (Fei and Midkiff, 2006; Mertz and Nunes, 2021) allow it to be given as a parameter. There are three approaches that adjust the sampling of the execution traces to cope with the monitoring performance overhead. Focusing on memory, Daoud et al. (2017) proposed a dynamic tracing approach that monitors memory usage to decide whether to collect a trace based on pre-defined conditions and thresholds such as the time elapsed since the last trace or the amount of memory allocation calls. Targeting the application workload, two approaches (Hauswirth and Chilimbi, 2004; Brönink and Rosenblum, 2016) propose to use a sampling rate that is inversely proportional to the frequency of execution, which gives the application throughput. Although this strategy is able to reduce the monitoring overhead when the application is overloaded, the collected sample may not correspond to the population of execution traces. In usage peaks, the proportion of collected execution traces is smaller than in typical workloads. As result, collected samples are likely not representative considering the population.

In summary, monitoring with a fixed configuration is suitable for software applications that have a workload with low variance, but may lead to an unacceptable overhead when there are peaks and cannot deal with unpredicted situations. Existing approaches dynamically adapt monitoring configurations by filtering traces or adjusting how the sampling is performed. The former assumes that there is a subset traces that are of interest and aim at collecting only them. The latter aims at having spacial coverage (that is, to monitor the whole application) but compromise temporal coverage by dynamically changing the sampling rate to keep the monitoring at an acceptable level. Our goal, similarly to Hauswirth and Chilimbi (2004) and Brönink and Rosenblum (2016), is to collect a sample of execution traces with spacial coverage. However, we propose a decision-making process to collect execution trace samples that are *representative* of the population. Our process is described in the next section, followed by an evaluation that compares our approach to their strategy (sampling rate inversely proportional to the throughput) and uniform sampling.

3. Adaptive sampling process

Given the limitations of existing work and our goal of collecting representative samples of detailed execution traces at runtime with controlled performance impact, we propose a three-activity process for monitoring software applications with an adaptive sampling rate. Our decisions are at the granularity of application request, which have method calls (executed to respond to it) recorded as detailed execution traces. This occurs if the request is selected to be part of a sample collected in a monitoring cycle. We use the *PetClinic*[2] project as a running example to explain the activities of our process. It is a web application that demonstrates the use of the Spring Framework and its features. It provides features (possible requests) in which employees of a pet clinic can view and manage information regarding veterinarians (`/vets`), clients (`/owners`), and their pets (`/pets`). It also includes a home page (`/home`), which is the entry point for users. We next first overview our process and its activities, and then describe each activity in detail.

3.1. Process overview

The key idea underlying our process is to decrease the monitoring overhead to an acceptable level when the target software application needs its resources for regular processing and increasing it after the situation has been normalized. At the same time, we keep track of general statistics about the sample and population of requests to identify when a sample is representative. Our monitoring process is performed in *cycles* and the result of each cycle is a representative sample. This behavior is shown in Fig. 1, where the black line represents the application workload, the red line represents the monitoring overhead, and the green line indicates the amount of requests and their execution traces being collected over time.

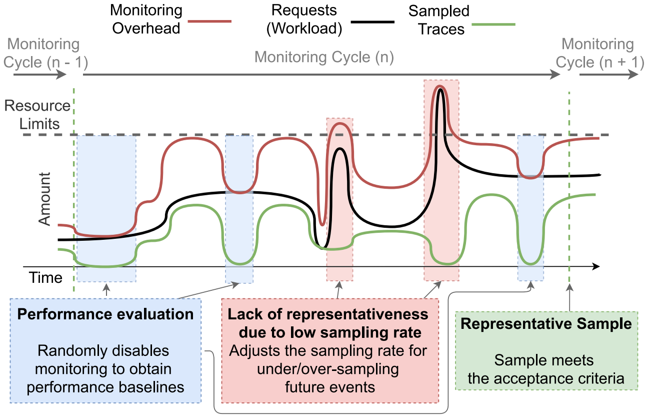


Fig. 1. Illustration of the Adaptive Sampling Process in Action: The figure shows how the sampling rate varies (in terms of the number of sampled traces) according to the current application workload. In peaks, the sampling rate is decreased to reduce the monitoring overhead. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

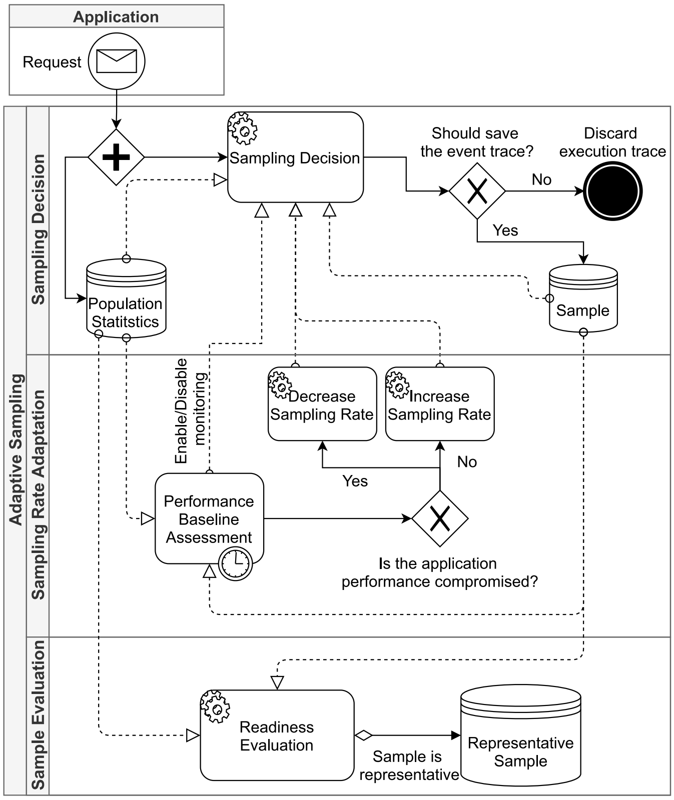


Fig. 2. Overview of our Adaptive Sampling Process.

In order to make this behavior possible, three activities are performed in parallel at runtime as part of our process, as shown in Fig. 2. The first activity, *Sampling Decision*, is responsible for deciding whether an application request should have its associated execution traces collected and stored in detail (which is costly and requires I/O), taking into account both the sampling rate and the representativeness of the sample compared to the population. The sample representativeness is based on the distribution of application requests. All requests go through screening and by doing so, we also keep general statistics of the population of requests.

[2] https://projects.spring.io/spring-petclinic/

The sampling rate used in the Sampling Decision activity is updated by the *Sampling Rate Adaptation* activity. It observes the current application workload as well as the monitoring overhead to decrease or increase the sampling rate. These adjustments of the sampling rate can be seen in Fig. 1—the larger the space between the black and red lines, the higher the sampling rate and, consequently, the monitoring overhead. When a performance degradation is perceived, i.e. the application response time increases, the sampling rate is decreased proportionally to the perceived degradation. To exemplify, suppose that an unexpected increase in the application workload results in the application bumping in the limits of its runtime platform (shadowed red boxes). In this situation, the monitoring is gradually reduced to allow the application to maintain its throughput. The sampling rate can be restored when the application workload decreases. To be able to assess the impact caused by monitoring, this activity also involves the collection of a performance baseline, which is a measurement of the application performance without monitoring. In Fig. 1, this occurs in the shadowed blue boxes.

Finally, the *Sample Evaluation* activity is responsible for continuously evaluating the sample being collected to identify when it is considered representative. When the sample satisfies acceptance criteria with respect to the representativeness of the population, it is released for analysis, and a new monitoring cycle starts. If our monitoring process is used, e.g., in a self-adaptive system, the analysis of the sample can take place right after each cycle is concluded. The rationale for using monitoring cycles is to pursue representativeness in shorter periods while considering a smaller population as a reference. This allows the approach to quickly react to sudden changes in the popularity of specific requests as a result of changes in the system workload and user's behavior. After broadly understanding how our process works, we next describe each of its activities.

3.2. Activity 1: Sampling decision

The *Sampling Decision* activity involves the execution of Algorithm 1 whenever a new observable request happens in the application. It performs three main tasks: (i) store statistics of the population of application requests (line 1); (ii) decide whether a request should be recorded with execution traces (lines 2–13) and (iii) store statistics of the request being added to the current sample (line 8), when applicable.

Storing statistics associated with the population and sample means keeping track of the *frequency distribution* of each request in these two sets. This information is used to decide whether a particular request should be recorded with execution traces or whether a sample is representative and the monitoring cycle is complete. By adding requests to the population and sample (lines 1 and 8), we keep the number of each possible request registered, as shown in Table 1 considering the PetClinic example. These values are obtained with an initial sampling rate of 50%.

A request is added to the sample and recorded with execution traces when three conditions are satisfied. The first refers to whether the monitoring is enabled (line 2). As introduced in the previous section, there are moments when the monitoring is disabled to obtain a performance baseline in terms of application response times—this is controlled by the Sampling Rate Adaptation activity. The second condition involves randomly deciding whether the request should be selected based on the current sampling rate (*rate*), which gives the probability of selecting a request as part of the sample. This decision is made using the Bernoulli distribution with parameter $p \in (0, 1)$ to assign the value 1 (*true*) with probability p and the value 0 (*false*) with probability $1-p$ to the *shouldSample* variable (line 3). The request satisfies the second criteria when *shouldSample* is *true*.

Algorithm 1: Sampling Decision

```
Input: request to be processed by the application;
Input: current sampling rate rate ∈ (0, 1);
Input: monitoring control isMonitoringEnabled;
Data: population, sample;
Result: True if the execution traces of request should be
        recorded, false otherwise.

1  add(request, population);
2  if isMonitoringEnabled then
3      shouldSample ← Bernoulli(rate);
4      if shouldSample then
5          $E_p \leftarrow \{x \in population \mid x = request.id\}$;
6          $E_s \leftarrow \{y \in sample \mid y = request.id\}$;
7          if $\left(\frac{|E_p|}{|population|} \geq \frac{|E_s|}{|sample|}, \epsilon\right)$ then
8              add(request, sample);
9              return true;
10         end
11     end
12 end
13 return false;
```

Table 1
Running example: Frequency distribution of the population and sample in the sampling decision activity.

Request	Population		Sample	
/home	105	(47.7%)	53	(47.7%)
/vets	43	(19.5%)	22	(19.8%)
/pets	62	(28.3%)	31	(27.9%)
/owners	10	(4.5%)	5	(4.5%)
Total	**220**	**(100%)**	**111**	**(55.5%)**

The third condition is related to the representativeness of the sample—we aim to keep its frequency distribution similar to that of the population (lines 5–7). The rationale is to not miss less frequent requests with, e.g., anomalies and exceptions. The idea is related to stratified sampling, where a population can be partitioned into subpopulations, i.e. clusters (Pirzadeh et al., 2011), and a representative sample has the same class distribution as the population. The verification that is performed consists of a runtime *resampling strategy* to balance the sample's class distribution according to the population's class distribution. This strategy is inspired by data analysis resampling to deal with unbalanced datasets (Estabrooks et al., 2004). By resampling, we balance the sample classes according to the population distribution to keep representativeness in terms of proportion. It consists of ignoring new requests from the majority classes to allow minority classes to include more requests and increase their cardinality. As previously said, we keep the statistics of the population (*population*) and sample (*sample*) (exemplified in Table 1). These are kept as key–value mappings, where the type of a request is the key (*request.id*), and its execution frequency within a monitoring cycle is the value. Using these statistics, it is possible to compute E_p and E_s, which are the amount of requests of a particular type in the *population* (line 5) and in the current *sample* (line 6), respectively. Based on these values, we test if the sample is lacking requests of the type in consideration, considering an error margin ϵ. For example, if a request of type /vets occurred (Table 1), it will be not added to the sample because it already has enough traces from /vets (19.8%) when compared to the population (19.5%). When a request satisfies this condition (line 7), the request is added to the sample (line 8) and the algorithm returns *true* (line

```
Algorithm 2: Sampling Rate Adaptation
 Input: current sample being collected in the monitoring cycle;
 Input: the maximum time window in which a performance baseline must be kept duration;
 Input: the current performance currentPerf;
 Input: monitoring control isMonitoringEnabled;
 Data: the current samplingRate; performanceRef;
 Result: The updated sampling rate.

1  addPerformanceSample(performanceRef, currentPerf, isMonitoringEnabled);
2  $medianRps \leftarrow \mathrm{median}(\{x \in performanceRef[RpS] \mid performanceRef[ME = isMonitoringEnabled]\})$;
3  $normalBehavior \leftarrow performanceRef[RpS = medianRpS \wedge ME = isMonitoringEnabled]$;
4  $equal \leftarrow \mathrm{ttest}(normalBehavior, currentPerf, 0.05)$;
5  $diff \leftarrow \frac{\sum_{i \in currentPerf} i}{\sum_{i \in normalBehavior} i} - 1$;
6  if isMonitoringEnabled then
7      if equal or diff > 0 then
8          $rate \leftarrow \min(rate + (rate * |diff|), maxRate)$;
9      else
10         enablePerformanceBaseline(duration);
11     end
12 else
13     if not equal and diff < 0 then
14         $rate \leftarrow \max(rate - (rate * |diff|), minRate)$;
15     end
16 end
17 return rate;
```

9), that is, the request should be recorded with execution traces. If any of the three conditions is not satisfied, the algorithm returns *false* and the request is not recorded, implying no additional monitoring overhead.

3.3. Activity 2: Sampling rate adaptation

The sampling rate used in the previously described activity is updated by Algorithm 2 executed in the Sampling Rate Adaptation activity. This is done considering the following premises.

1. A software engineer is able to provide a desired sampling rate that leads to an *acceptable performance impact* caused by monitoring when the application *workload is typical*.
2. The acceptable performance impact is not in terms of percentage but the absolute increase in the response time. For example, if the response time of a request is typically 100 ms and with monitoring 105 ms, the acceptable performance impact is 5 ms and not 5% of overhead.
3. The sampling rate should be reduced to prevent an increase in the response time when the application is under stress, limited by a minimum required sampling rate.
4. The sampling rate should be increased if it is below the desired level and the software application is returning to its typical behavior after a peak.

Based on these premises, the adaptation of the sampling rate requires three inputs: (i) *maxRate*, which is the desired sampling rate and a higher sampling rate is not needed; (ii) *minRate*, which is the minimum required sampling rate; (iii) the *duration* of the period in which the approach should collect data to understand the application performance with the monitoring disabled (this is required for assessing the monitoring performance impact); and (iv) the *frequency* in which the sampling rate is revised (in seconds).

Table 2
Running example: *performanceRef* table containing the response time of each request according to a given workload in requests per second (RpS). The ME column indicates whether the record was collected when monitoring was enabled. Rows highlighted in gray refer to the median of each group (ME = true and ME = false).

#	RpS	/home	/vets	/pets	/owners	ME
1	500	325 ms	450 ms	800 ms	1200 ms	true
2	1500	400 ms	550 ms	900 ms	1500 ms	true
3	2500	600 ms	780 ms	1050 ms	1100 ms	false
4	550	350 ms	400 ms	650 ms	900 ms	true
...	...	...	...	...	...	...
$n-1$	325	430 ms	420 ms	480 ms	700 ms	false
n	200	270 ms	200 ms	235 ms	500 ms	true

Algorithm 2 performs the following tasks. First, it stores statistics of the application performance (line 1). The parameter *currentPerf* gives the current application performance as a record with the number of requests executed since the last algorithm execution (RpS) and the average response time of each executed request. These data are stored in the *performanceRef* table, which consists of a performance sample. For each record, we also store a flag indicating if these data correspond to a period in which monitoring is enabled. An example of *performanceRef* is shown in Table 2 for PetClinic. To avoid bias towards past observations and consuming unnecessary resources, *performanceRef* is size-limited and stores the most recent executions, i.e. the oldest record is discarded to store a new one when the size limit is reached.

Based on these statistics, it is possible to derive both the typical application workload *medianRpS* (line 2) and the corresponding normal behavior *normalBehavior* in terms of response time (line 3), with and without monitoring. The typical application workload is given by the median of requests per second *medianRpS* of the records in *performanceRef*. Considering our example, these are the records #2 and #3, with and without monitoring, respectively. Because we need to select a single record to be used in the next algorithm tasks, if *performanceRef* contains an even number of records, we select the highest requests per second (as opposed to the arithmetic mean of the two middle values) and the associated record, which is the *normalBehavior*.

Then, we test if the current application performance *currentPerf* is significantly different from the normal behavior *normalBehavior* of the application to detect performance variations (line 4). This is done by comparing the averages of the execution times of each request type with a paired t-test, with the null hypothesis that the mean of the paired differences between *currentPerf* and *normalBehavior* is 0, with a significance level of 95% ($p = 0.05$). Assume that Algorithm 2 is executing in our example with monitoring disabled and thus the *normalBehavior* is record #3. Let *currentPerf* be

$$\{\langle /home, 500\rangle, \langle /vets, 720\rangle, \langle /pets, 950\rangle, \langle /owners, 1020\rangle\}. \quad (1)$$

The result of the comparison of *normalBehavior* and *currentPerf* is assigned to the variable *equal*, which is the result of

$$ttest(\langle 600, 780, 1050, 1100\rangle, \langle 500, 720, 950, 1020\rangle, 0.05). \quad (2)$$

This results in $equal = true$, indicating that there is no significant difference between these two groups. This is the first indicator of whether the sampling rate should be updated. The second indicator is the difference *diff* (in percentage) between *currentPerf* and *normalBehavior* (line 5). In the example, it is

$$diff = (3190/3530) - 1 = 0.9037 - 1 = -0.0963 \quad (3)$$

This means that the current performance is 9.63% lower than the normal behavior.

These two indicators (*equal* and *diff*) are used together with the monitoring state of the application to decide whether the sampling rate should be updated. The sampling rate is increased – proportionally to the observed difference, limited by *maxRate* – if the current performance (which includes monitoring) is similar or better than the normal behavior (lines 7–8). This means that the current execution times of the requests are similar or faster than the past observations, and thus the monitoring overhead is acceptable. The sampling rate is decreased (also based on *diff*), if the normal behavior is significantly different from the current performance (which does not include monitoring, limited by *minRate*) and the current performance is worse than the normal behavior (lines 13–15). This case occurs when the current execution times of the requests are slower when compared to past observations. In this situation, the application is (a) being impacted by users with an increased workload, or (b) the monitoring overhead is impacting the performance above acceptable levels. Note that $minRate > 0$ because, if the sampling rate reaches 0, it remains 0 indefinitely.

In order to identify whether the application performance still faces degradation regardless of the case, there is a need for collecting a performance baseline (line 10). This occurs when the application is being monitored and the current performance is worse (significantly different and lower) than the normal behavior. In this case, the monitoring is then globally disabled and the performance baseline is collected according to the specified *duration*. Finally, if none of these conditions are met, the sampling rate remains the same.

3.4. Activity 3: Sample evaluation

Our process aims at collecting a representative sample in each monitoring cycle. Therefore, when a new request is added to the sample, the sample representativeness is evaluated to determine if it is ready for being used. This is done in the Sample Evaluation activity that involves the execution of Algorithm 3. A sample is considered representative if it satisfies three criteria: (i) it is larger than the minimum sample size; (ii) the performance between the sample and population is equivalent in terms of response times; and (iii) the sample distribution is similar to the population distribution. When the sample satisfies these criteria, there is statistical evidence that it is representative.

To prevent a scenario in which the sample being collected is never representative, Algorithm 3 uses an exponential *decaying confidence* (line 1). It is used to adjust the required level of representativeness of the sample based on the length of the monitoring cycle. The confidence (z) starts at 100% and decreases by a constant λ every second (t) in the monitoring cycle, where $\lambda = 1/maxLength$. *maxLength* is the maximum length of the monitoring cycle, which is the required parameter of this activity.

The first criterion – sample size – is checked in line 4. The required sample size (line 3) is given by Cochran's minimum sample size (Cochran, 1977) (line 2), with finite population correction, where the decaying confidence level z is used to estimate the associated standard normal distribution (e.g., 1.96 for $z = 0.95$), e is the margin of error ($e = 0.05$) and p is the degree of variability of the population, indicating how heterogeneous the population is. We use $p = 0.5$ because it does not assume that the population is homogeneous and leads to higher samples, being thus a conservative choice.

Given that we track the performance of application requests in terms of average response times, we test the equivalence between the sample and population – stored in Table 2 – using this measurement, which is the second criterion. This is done by verifying if the frequency distribution of the sample significantly differs from the postulated population mean using a one-sample parametric t-test (two-sided) (line 5) to test the null hypothesis that the sample mean is equal to the population mean, with a decaying confidence level (computed in line 1). The test compares the average values of the two data sets and determines if they came from the same population.

Algorithm 3: Sample Evaluation

Input: the length of the current monitoring cycle t in seconds;
Data: the current *population* and *sample*;
Result: *true* is the *sample* is representative of the *population*.

```
1  $z \leftarrow 100 * e^{-\lambda t}$;
2  $n_\infty \leftarrow \frac{z^2 p(1-p)}{e^2}$;
3  $n \leftarrow \frac{n_\infty}{1 + \frac{n_\infty - 1}{|population|}}$;
4  if $|sample| > n$ then
5      if t-test(population, sample, z) then
6          balanced ← true;
7          foreach request in population do
8              $E_p \leftarrow \{x \in population \mid x.id = request.id\}$;
9              $E_s \leftarrow \{y \in sample \mid y.id = request.id\}$;
10             if not $\left(\frac{|E_p|}{|population|} = \frac{|E_s|}{|sample|}, z\right)$ then
11                 balanced ← false;
12             end
13         end
14         if balanced then
15             release sample for analysis;
16             $sample \leftarrow \emptyset$;
17             $population \leftarrow \emptyset$;
18             return true;
19         end
20     end
21 end
22 return false;
```

Finally, the third criterion is evaluated in lines 6–14, which checks the balance in the request distribution, measuring the over- or under-representation of requests in the sample compared to the population. This is similar to the comparison shown in Table 1. However, in line 10, the comparison between each request type in the sample and in the population considers a margin of error of z (decaying confidence level). This evaluation aims to ensure that the sample is balanced according to the population to give confidence that all requests, even those that rarely happen, are present.

When the sample meets all these three criteria, it is said representative and is released for analysis. After releasing the sample for analysis, we reset the sample and population (lines 16 and 17), and a new monitoring cycle starts.

4. Evaluation

Having described our monitoring process in detail, we now evaluate it using applications of a widely known and used benchmark.

4.1. Evaluation settings

4.1.1. Research questions and metrics

Our monitoring process aims at collecting representative samples of execution traces and, at the same time, keeping the performance overhead at an acceptable level. Therefore, the goal of evaluation is to assess these two aspects, which are the focus of our two research questions, listed as follows. Metrics used to answer each question are also detailed.

RQ1 What is the *performance impact* of our monitoring process?

TR Throughput (average number of requests/second)

SR Average sampling rate/second

RQ2 What is the *representativeness* of the samples of executions traces collected with our monitoring process?

RMSE Root-mean-square error of memory consumption

Whenever we monitor a software application, there is performance overhead. Therefore, in RQ1, we quantitatively assess this overhead by measuring the application throughput (TR). In addition, given that the monitoring overhead is proportional to the sampling rate, to better understand the throughput, we also measure the sampling rate (SR).

A dynamic sampling rate may have a negative effect on the representativeness of collected traces, making them less useful for understanding the application behavior for a particular goal, such as debugging. For not biasing our evaluation towards our approach, in RQ2 we do not use any criteria on which our monitoring process relies to assess sample representativeness (e.g. response times). Instead, we use a software characteristic – memory consumed by each request served by the application in terms of the average of memory consumed by methods executed due to the request processing – that can be understood through monitoring, is straightforward to be collected, and can have its correctness evaluated. The measurement of memory consumed by methods may not be trivially obtained in some execution platforms. However, there is a commonly used way of obtaining this measurement, i.e. computing the difference between the available memory before and after the method execution. Such measurement is enough to compare two different requests executed by the application in terms of memory footprint (i.e. the average amount of memory used by each method called due to the request processing), considering that the workload variations and environment configuration are exactly the same for both executions. As we are pursuing a representative sample of execution traces, it is expected that the memory footprint resulting from the collected traces is representative of the population (i.e. all the methods processed by the application). Thus, based on a ground truth (explained as follows), we can assess the error (RMSE) of the sample with respect to the average memory usage of each method. If RMSE is low, it indicates that the collected traces are reliable for debugging memory consumption, for example.

4.1.2. Compared approaches

Our adaptive monitoring process (ADP) is compared to the main alternative approach (Hauswirth and Chilimbi, 2004; Brönink and Rosenblum, 2016), which is a sampling rate inversely proportional to the workload given by the throughput (INV), and a practically used approach, which is uniform sampling (UNI). The selected uniform sampling rate is 50%, which is the same used as initial (and desired) sampling rate for ADP and INV. As a reference, we also execute our evaluation with two additional configurations. The first is no monitoring (NOM), in which there is no monitoring overhead and it thus provides the maximum (best) possible value for throughput. The second is full monitoring (FUM), in which every application request is recorded with execution traces. FUM gives thus the minimum (worst) possible throughput value and also serves as ground truth for calculating RMSE because it contains the memory measurements of all executions of each request made during a workload simulation (i.e. the population). The evaluation was designed and implemented in such a way that the different sampling strategies collect only the information they need. Due to its sophisticated decision process, ADP needs to collect and maintain more information about the application execution, such as throughput, response times, cycle length, and sample/population metrics. INV demands only throughput, and UNI does not need any information as it is a static and pre-defined sampling strategy. Regardless of sampling strategy, when an application request is selected to be sampled during the evaluation, the exact same amount of information and details about the method traces involved in the request are collected.

4.1.3. Target applications

We use the DaCapo (Blackburn et al., 2006b,a) benchmark suite for our evaluation. DaCapo provides various Java applications to evaluate approaches that focus on the execution environment of applications. Consequently, it does not explicitly provide extension points for customizing its execution. Because we need to simulate a workload with variation (not simply firing requests one after another) as well as instrument the applications, we selected a subset of five applications to be instrumented, described as follows.

`cassandra` Executes queries to recover documents from the NoSQL database management system Cassandra.

`h2` Executes SQL transactions against a model of a banking application on top of the H2 database.

`lusearch` Executes search queries against the document search engine Lucene.

`h2` Executes SQL transactions against a model of a banking application on top of the H2 database.

`lusearch` Executes search queries against the document search engine Lucene.

`tradebeans` Runs HTTP requests via Java Beans against a web application that simulates a stock trading system.

`xalan` Calls multiple times an XSLT processor for transforming XML documents into HTML pages.

The rationale for selecting these particular applications is that they are all distributed applications designed to process multiple requests in parallel, e.g. web requests or database queries, and are based on domains in which monitoring is valuable and difficult to control in terms of overhead.

4.1.4. Procedure

We instrumented all the target applications (which are open source) to enable monitoring using aspect orientation. Aspects intercept method calls generated by any request performed to the application, and then there are four Java implementations (ADP, INV, UNI, and FUM) to decide whether an application request should be recorded and, if so, collect all the execution traces generated by that request to compute our metrics such as the average memory consumption of methods executed and response time. NOM corresponds to the original version of the applications.

As introduced, our monitoring process requires a set of parameters. They consist of a single configuration for all target applications and not values that must be tuned for specific applications. The Sampling Rate Adaptation activity is triggered every 1 s and the collection of performance baselines last 3 s. The maximum and minimum sampling rates are 50% (as in UNI) and 1%, respectively. Finally, the maximum length of a monitoring cycle is 180 s (3 min).

For generating application workloads, we use the workload simulation provided by DaCapo, which relies on a navigation pattern that either falls into a specific distribution (transition table) or follows a specific sequence of executions. We use this navigation pattern to simulate a varying number of simultaneous users (i.e. threads). This allows us to observe the impact of monitoring when the application is under various stress conditions and how the sampling rate is adjusted. Inspired by load intensity modeling approaches (Von Kistowski et al., 2017), the designed workload includes (a) situations in which it keeps a stationary number of users, (b) seasonal patterns, and (c) bursts in the number of simultaneous users. Therefore, the resulting workload covers the typical variations regarding usage scenarios that may occur in production environments. The same workload settings are used to execute each application with each compared approach.

The simulations were executed on an Intel i7 2 GHz with 16G RAM. The maximum heap size of the Java Virtual Machine (JVM) was limited to 4 GB to cause the applications to execute under stress (with limited resources considering the workload). Each simulation was executed 10 times. The maximum number of simultaneous users was selected based on the identification of which number of users causes the application to deteriorate its performance due to the lack of resources. With 4 GB of RAM available, this number varies from 6 to 200. The measurement used to compare application executions and answer RQ2 – memory consumed by methods – is not explicitly made available by the JVM. Thus, we use the discussed standard way to derive it, i.e. computing the difference between the available memory before and after the method execution. Given that there are moments in which JVM garbage collector is executed to free up memory, some of the collected measurements are negative and thus invalid. These are, therefore, discarded.

The error used in RMSE in our evaluation is given by the difference between the average memory consumption (in kilobytes) of all executions of a particular request (obtained with FUM) and the average memory consumption of executions monitored by a compared approach (ADP, INV, and UNI) of this request. RMSE is thus calculated as follows.

$$RMSE = \sqrt{\frac{\sum_{rinReq}(\mu_{FUN} - \mu_S)^2}{|Req|}}$$

where Req is the set of possible requests, μ_{FUN} is the average of memory consumption of all executions of request r (measured by μ_{FUN}), μ_S is the average of memory consumption of executions of request r measured by the sampling approach $S \in \{ADP, INV, UNI\}$, and $|Req|$ is the number of possible requests.

4.2. Results

The results obtained following the procedure described above are presented in Table 3. It shows the values obtained for each metric (TR, SR, and RMSE) with each compared approach (ADP, INV, UNI) and reference values (NOM and FUM) for each target application. Because we run the simulation 10 times for each configuration, we present the mean and standard deviation. As can be seen, the results are consistent across all applications, even though the applications vary in nature. A Friedman's test showed that there is significant difference among the compared approaches, both for TR ($\chi^2(2) = 84.28$, $p < 0.001$) and RMSE ($\chi^2(2) = 74.68$, $p < 0.001$). Post-hoc analysis with pairwise comparisons using Nemenyi-Wilcoxon-Wilcox all-pairs test for a two-way balanced complete block design revealed that this is due to the differences among all approaches in both cases.

With respect to performance overhead (RQ1), as expected, UNI achieves the worst results, causing the throughput to be 13.5–23.1% lower than NOM. The impact is approximately 50% of FUM, as it collects execution traces of roughly half of the requests. INV has the lowest performance overhead, with an overhead ranging from 4.6% to 11.6%. This occurs because it always reduces the sampling rate with more intense workloads, regardless of its impact on the collected execution traces. ADP, in turn, is the "middle option", which has an overhead from 10.3% to 14.2%, as it also takes sample representativeness into account while monitoring the application.

The performance overhead is in accordance with the sampling rate. The lower the performance overhead, the lower the sampling rate. Despite achieving the intermediate results, in all cases, ADP has a performance overhead closer to INV than to UNI. Nevertheless, its average sampling rate is, also in all cases, closer to UNI than to INV, sometimes as high as 48.7% (note that, by configuration, the sampling rate is always limited to 50%). This indicates that ADP is able to choose the moments in which the sampling rate should be reduced (this is further discussed in the next section) as well as to reduce the sampling rate in a sustainable manner.

With respect to the error present in collected samples (RQ2), ADP is not the middle option. In all cases, it has the lowest (best) results for RMSE. This provides evidence that ADP is able to collect execution traces that better represent the population. Although, as expected, INV has the highest error for most applications, this is not the case for tradebeans. A possible explanation is that the memory consumption of the different application requests largely varies for this application and, in this particular case, relying on randomization to collect traces, even with higher sampling rates, cannot guarantee good results.

Conclusion. ADP is able to collect the most representative samples of execution traces, using memory consumption as representativeness measure. The error of the collected samples is 9–54% and 12–44% lower than INV and UNI, respectively. It also significantly reduces the performance overhead of UNI (3–12% lower). Although it has a performance overhead higher than INV, it is much lower (1–6%) than the reduction of the error in the collected samples.

4.3. Detailed analysis

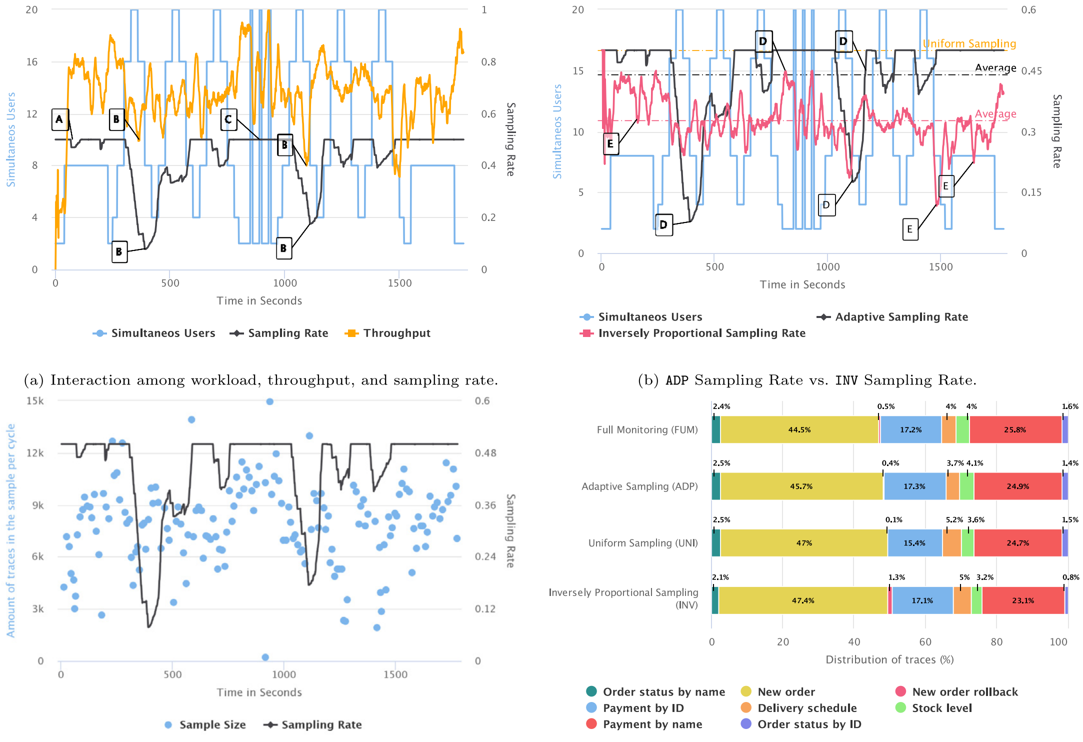
In the previous section, the results show that ADP significantly improves the representativeness of the collected samples of execution traces, with little impact on the performance overhead. To explain these results, we analyze in detail the results obtained with h2, shown in Fig. 3.[3] From the 10 executions, we selected that with the median throughput value.

Interaction among workload, throughput, and sampling rate. We first analyze what happens over the course of the simulation in Fig. 3(a). The blue line shows the application workload. The typical workload is 8 simultaneous users, which can be 20 in peeks (recall that the memory limit is 4 GB, which causes the

[3] All applications have similar results. Due to space restrictions, their charts are available in our complementary material at https://www.inf.ufrgs.br/prosoft/resources/2022/jss-adaptive-sampling.

Table 3
Simulation Results: Comparison of the values obtained for the metrics Throughput (TR), Sampling Rate (SR), and Root-mean Square Error (RMSE).

	Monitoring	Throughput	Sampling rate	Root-mean square error
cassandra	NOM	23180.4 ± 488.6	0%	–
	FUM	16769.5 ± 468.7 (−27.6%)	100.0%	–
	ADP	20763.9 ± 666.2 (−10.4%)	48.7%±1.3	496.6 ± 42.9
	INV	21112.7 ± 520.6 (−8.9%)	29.2%±2.5	699.9 ± 57.0
	UNI	18900.2 ± 661.7 (−18.4%)	50.0%	651.6 ± 75.2
h2	NOM	1829.0 ± 20.1	0%	–
	FUM	1197.1 ± 30.2 (−34.5%)	100.0%	–
	ADP	1587.1 ± 19.8 (−13.2%)	44.0%±3.4	628.0 ± 108.6
	INV	1633.2 ± 24.4 (−10.7%)	32.7%±1.5	1291.9 ± 118.7
	UNI	1517.0 ± 19.1 (−17.0%)	50.0%	1196.1 ± 102.8
lusearch	NOM	74376.7 ± 213.8	0%	–
	FUM	49397.7 ± 363.9 (−33.5%)	100.0%	–
	ADP	66267.2 ± 324.2 (−10.9%)	41.6%±2.2	1394.2 ± 349.0
	INV	70951.0 ± 216.0 (−4.6%)	28.0%±1.6	3010.8 ± 450.0
	UNI	57142.0 ± 433.3 (−23.1%)	50.0%	2086.0 ± 251.1
tradebeans	NOM	1832.2 ± 20.2	0%	–
	FUM	1204.6 ± 17.9 (−34.2%)	100.0%	–
	ADP	1571.8 ± 14.0 (−14.2%)	48.4%±0.9	721.2 ± 40.3
	INV	1619.6 ± 15.1 (−11.6%)	32.7%±1.1	797.7 ± 55.5
	UNI	1512.8 ± 22.6 (−17.4%)	50.0%	831.0 ± 36.7
xalan	NOM	266.8 ± 1.0	0%	–
	FUM	182.6 ± 1.7 (−31.5%)	100.0%	–
	ADP	239.1 ± 1.5 (−10.3%)	47.5%±0.9	111.7 ± 9.1
	INV	241.1 ± 1.7 (−9.6%)	24.0%±0.6	187.4 ± 13.2
	UNI	230.5 ± 1.3 (−13.5%)	50.0%	135.0 ± 10.3



(a) Interaction among workload, throughput, and sampling rate.

(b) ADP Sampling Rate vs. INV Sampling Rate.

(c) Collected sample sizes over time.

(d) Comparison of the distribution of application requests.

Fig. 3. Analysis of the ADP Results with the h2 Application. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

application to be under stress with a relatively low number of users). As explained, the workload has stationary segments, seasonal patterns and bursts. In stationary segments, ADP is able to keep the sampling rate at a value close to 50% (label A), with small decreases due to variances in the response time, as this is the metric used to adapt the sampling rate. In seasonal patterns, ADP detects performance degradation and reduces the sampling rate (label B). Note that even with a decreased sampling rate, the throughput (orange line) decreases, showing that the user requests are causing the application to be under stress. Lastly, in isolated bursts, the sampling rate remains at 50% (label C) because the increased number of users for brief moments does not have a major impact on the application performance. As can be seen, despite the monitoring and the peeks, the throughput is not lower than in the rest of the simulation.

ADP sampling rate vs. INV sampling rate. Now we look in detail at the sampling rate controlled by INV and how it differs from ADP. Both approaches apply mechanisms to reduce the sampling rate when the application is struggling with an increased workload (label D). However, while ADP uses the response time to make decisions, INV relies on the workload (throughput). In many cases, this correctly reduces the sampling rate to not cause a major performance impact on the application. But in certain situations (label E), low throughput is due to a low number of requests, thus there is no need to reduce the sampling rate. ADP is able to better understand the application as a whole as it keeps track of a performance baseline with and without monitoring, allowing it to identify when the monitoring is competing for resources with the application.

Collected sample sizes. ADP does not focus on collecting execution traces to be analyzed all together, but works in cycles providing a set of samples of execution traces, each being representative of the population in each cycle. Although there is a timeout for cycles, ideally the cycle ends when the representativeness criteria are met, leading to samples of various sizes. We present the collected samples sizes for h2 in Fig. 3(c). The horizontal proximity between the dots indicates that no cycle reached the timeout of 180 s—the maximum cycle time is 25 s. Fig. 3(c) also shows that the sample size is not correlated to the sampling rate. This may occur in h2 due to the low number of types of requests (8 distinct types) because it is easier to have similar distributions when the number of classes to be compared between the sample and population is low. Note that there is an outlier cycle composed only of 211 execution traces and that lasted less than 1 s. This indicates that the sample satisfied the representativeness criteria with high confidence because the longer the cycle, the lower the confidence level as it decays over time. As result, on average, h2 had 163 cycles. Because the request types of xalan and tradebeans are also low, 16 and 12, respectively, it is possible to collect representative samples in shorter times, resulting in 422 and 151 cycles, respectively. cassandra and lusearch, in turn, have more than 100 request types, causing the lowest number of cycles (122 and 48, respectively). We observed longer monitoring cycles in these two applications, including timeouts.

Distribution of application requests. We now analyze the distribution of application requests in Fig. 3(d) considering the data in all samples collected during one execution of the simulation of h2. FUM shows the distribution of the population (ground truth). Although ADP checks for distribution similarity by sample, the resulting set of samples has the distribution most similar to the population (considering the whole simulation), having the new order request the highest difference (1.2%). This request type, which is the most frequent, also led to the highest difference for UNI and INV. UNI has a difference of 2.5%; while INV, which focuses on performance rather than representativeness, has the highest difference (2.9%) among the three approaches.

Variable vs. fixed-length monitoring cycle. The usage of variable-length cycles is inherent to ADP as it was designed to actively pursue representativeness in shorter periods, allowing it to quickly react to sudden changes in the system workload. However, as we use the dynamic cycles identified by ADP to establish a time window comparison with baselines in Table 3, it may represent a bias in favor of ADP. Thus, we now analyze the implications of using a variable vs. fixed-length monitoring cycle for INV and UNI in terms of its impact on the collected samples. Table 4 presents the results of the evaluation using fixed-length monitoring cycles. As the cycle length timeout used by ADP was 3 min, we evaluated using different values for comparison. With respect to the error present in collected samples, the computed RMSE from INV and UNI when compared to FUM for all the evaluated fixed cycle lengths always fall within the upper or lower bounds when considering the standard deviation and comparing with values achieved with variable cycle length identified by ADP. Fig. 4 presents the same results as Table 4, but focuses on h2. This provides evidence that the cycle length does not significantly impact on how INV and UNI collect samples, which can be explained by the fact that in these approaches the sampling rate and sampling decision are not influenced by the size of the sample or by the time window in which the sample is being collected. Thus, as the benchmarks used tend to apply the same workload variations for all the executions, it results in similar traces being collected.

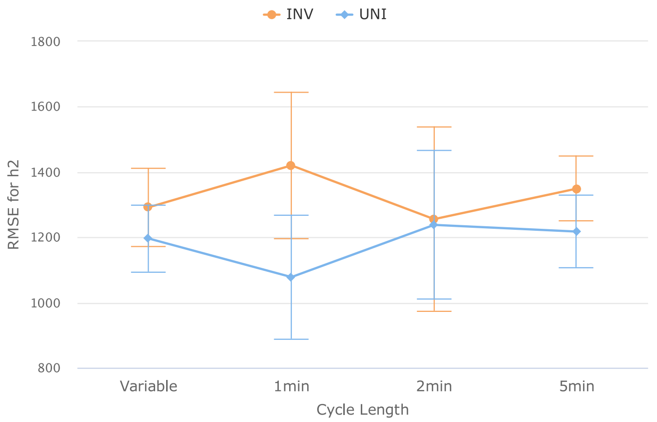


Fig. 4. Comparison of Root-mean Square Error (RMSE) values obtained from INV and UNI for h2 benchmark when using a variable monitoring cycle length (identified by ADP) vs. fixed-length monitoring cycles of 1, 2 or 5 min.

4.4. Threats to validity

Our evaluation involves runtime execution with a particular workload and, thus, there are many settings that may influence the results. All our settings were selected to avoid bias. The fired application requests have a key role in the obtained results. To minimize the chance of using a workload that favors a particular approach, we rely on the randomness and reliability provided by DaCapo. Another workload configuration that may influence the results is the number of simultaneous users and how it varies over time. Our designed workload includes different types of variations, which are those used in previous work. Moreover, the maximum number of users, based on preliminary executions, was selected to guarantee that the application executes under stress at certain moments.

Another construction threat to validity is how we assess representativeness. The key goal is to evaluate whether the desired execution traces are included in the sample. Given that this depends on the monitoring goal, we use memory usage due to the reasons explained in the study settings. This measurement is not used by any of the compared approaches for adapting the sampling rate or making decisions. The only challenge is to collect this information in Java, because its virtual machine offers limited support to fine-grained memory measurements and, in addition, it has multiple features that can affect this kind of measurement

Table 4
Comparison of Root-mean Square Error (RMSE) values obtained from INV and UNI when using a variable monitoring cycle length (identified by ADP) vs. fixed-length monitoring cycles of 1, 2 or 5 min.

Monitoring	RMSE			
	Variable-length cycles	1-min cycles	2-min cycles	5-min cycles
cassandra				
INV	699.9 ± 57.0	678.6 ± 81.5	724.4 ± 60.1	709.4 ± 34.0
UNI	651.6 ± 75.2	542.2 ± 95.0	580.8 ± 72.0	619.4 ± 45.7
h2				
INV	1291.9 ± 118.7	1420.2 ± 225.1	1255.5 ± 282.3	1349.0 ± 98.8
UNI	1196.1 ± 102.8	1077.9 ± 189.7	1237.9 ± 226.9	1217.4 ± 111.0
lusearch				
INV	3010.8 ± 450.0	3372.6 ± 664.8	3914.4 ± 523.7	3704.6 ± 335.6
UNI	2086.0 ± 251.1	2341.9 ± 540.5	2110.4 ± 276.3	1737.0 ± 221.5
tradebeans				
INV	797.7 ± 55.5	961.2 ± 184.8	838.5 ± 153.2	768.0 ± 169.7
UNI	831.0 ± 36.7	892.9 ± 104.5	876.7 ± 109.7	833.9 ± 91.9
xalan				
INV	187.4 ± 13.2	156.2 ± 43.1	195.0 ± 22.0	173.5 ± 19.3
UNI	135.0 ± 10.3	134.9 ± 38.8	149.7 ± 14.8	136.5 ± 10.9

during the application execution, such as the garbage collector and just-in-time compilation. We used a standard way to measure memory usage as well as discarded invalid measurements – negative values due to the execution of the garbage collector – for all approaches, including FUM.

In addition, our evaluation regarding the achieved representativeness of the collected sample is limited to the memory consumption and distribution of traces. Although these metrics are generic and not biased toward specific approaches, scenarios and workloads, the usage of other metrics would contribute to the generalization of the results. However, based on our empirical evaluation, we acknowledge that any metric used to assess the representativeness as part of an evaluation must be independent of a specific monitoring goal (e.g. identification of performance improvement opportunities or root causes of request errors). Otherwise, creating synthetic unbiased scenarios and workloads to assess such specific monitoring goals becomes complex.

Finally, an external threat to validity is the set of target applications. We selected applications that use various technologies, are of different domains, and vary in processing nature, e.g. while some make extensive memory usage, others rely on I/O or processor. Though the number of applications is not large, we emphasize that the obtained results are consistent across all applications and thus provide evidence of the generalization of the results. Yet, as any empirical study, further evaluations with different settings would improve the generality and reliability of the results.

4.5. Limitations

We now point out the limitations of our monitoring process. A monitoring cycle finishes when the representativeness criteria are met. In situations where an application must timely react to particular requests, this may cause the application to give a delayed response. We addressed this issue using a decaying confidence level based on the monitoring cycle time frame, which can be customized. However, the more elapsed time, the lower the confidence level. Therefore, if an application requires samples with some confidence level guarantees, the sample evaluation activity must be adapted.

In our work, we monitor applications by continuously making decisions and adaptations to collect execution traces, which implies an overhead higher than making simple adjustments (as in INV). This is, however, done in a lightweight way and our evaluation showed that despite the execution of our process activities, we obtain the most representative samples with a performance not far from INV. Yet, it is possible to reduce the cost of the Sampling Rate Adaptation activity by using bootstrapping and other statistical techniques to generate data from samples and estimate the population based on monitoring time frames instead of instrumenting all the requests. Then, the monitoring can be disabled for extended periods when the sample is in good shape to be used in bootstrapping.

5. Conclusion

Software runtime monitoring has been largely used for a wide range of purposes, from debugging to self-adaptation. When it collects costly information like detailed execution traces in production environments, it is crucial to prevent the monitoring to cause unacceptable overhead. A typical approach is thus to sample the traces. However, it is important to pursue that the collected traces are representative of the population of execution traces.

In this paper, we proposed a monitoring process to find the sweet spot between these conflicting goals, i.e. overhead vs. representativeness. Our process is performed in monitoring cycles and is composed of three activities, which use algorithms with statistical foundations to decide whether a particular application request must be recorded, when and to what degree adapt the sampling rate, and determine when a sample has been collected to, then, begin a new monitoring cycle. We evaluated our process by comparing it with monitoring performed with uniform sampling and a sampling rate that is inversely proportional to the workload (INV) as well as used executions with no monitoring and monitoring every application request as a reference. Our results show that our approach collects samples with the lowest errors with respect to the population, having a performance impact that is only 1%–6% higher than INV, which achieves the highest error rate.

Our approach is language independent. However, it was implemented in Java for our empirical evaluation. Our future work involves implementing and evaluating our approach in projects written in other programming languages. In addition, as monitoring can be used for different purposes, we aim to assess the effectiveness of our monitoring process for post-mortem fault analysis. Finally, we aim to adapt the process for distributed architectures.

CRediT authorship contribution statement

Jhonny Mertz: Conceptualization, Methodology, Investigation, Software, Validation, Writing – original draft, Visualization. **Ingrid Nunes:** Conceptualization, Methodology, Writing – original draft, Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

A link to data/code is included in the paper, under section "Evaluation", as a footnote.

Acknowledgments

Ingrid Nunes thanks for CNPq, Brazil grants ref. 313357/2018-8 and ref. 428157/2018-1. This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001.

References

Apiwattanapong, T., Harrold, M.J., 2002. Selective path profiling. In: ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, No. 1. ACM Press, Charleston, SC, USA, pp. 35–42. http://dx.doi.org/10.1145/634636.586104, URL: https://dl.acm.org/doi/abs/10.1145/634636.586104.

Bartocci, E., Falcone, Y., Francalanza, A., Reger, G., 2018. Introduction to Runtime Verification. In: Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 10457 LNCS, Springer, Cham, NA, pp. 1–33. http://dx.doi.org/10.1007/978-3-319-75632-5_1.

Blackburn, S.M., Garner, R., Hoffman, C., Khan, A.M., McKinley, K.S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S.Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J.E.B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., Wiedermann, B., 2006a. The DaCapo Benchmarks: Java Benchmarking Development and Analysis (Extended Version). Technical Report TR-CS-06-01, ANU, http://www.dacapobench.org.

Blackburn, S.M., Garner, R., Hoffman, C., Khan, A.M., McKinley, K.S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S.Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J.E.B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., Wiedermann, B., 2006b. The DaCapo benchmarks: Java benchmarking development and analysis. In: OOPSLA '06: Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications. ACM Press, New York, NY, USA, pp. 169–190, http://doi.acm.org/10.1145/1167473.1167488.

Brönink, M., Rosenblum, D.S., 2016. Mining performance specifications. In: Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Vol. 13-18-Nove. Association for Computing Machinery, New York, New York, USA, pp. 39–49. http://dx.doi.org/10.1145/2950290.2950314, URL: http://dl.acm.org/citation.cfm?doid=2950290.2950314.

Cassar, I., Francalanza, A., Aceto, L., Ingólfsdóttir, A., 2017. A survey of runtime monitoring instrumentation techniques. In: Francalanza, A., Pace, G.J. (Eds.), Proceedings Second International Workshop on Pre- and Post-Deployment Verification Techniques. PrePost@IFM 2017, Torino, Italy, 19 September 2017, In: EPTCS, vol. 254, Springer, Cham, Torino, Italy, pp. 15–28. http://dx.doi.org/10.4204/EPTCS.254.2.

Chan, A., Holmes, R., Murphy, G.C., Ying, A.T., 2003. Scaling an object-oriented system execution visualizer through sampling. In: Proceedings - IEEE Workshop on Program Comprehension, Vol. 2003-May. IEEE Comput. Soc, Portland, Oregon, USA, pp. 237–244. http://dx.doi.org/10.1109/WPC.2003.1199207.

Cochran, W.G., 1977. Sampling Techniques. John Wiley & Sons, Ltd, New York, New York, USA, pp. 89–149.

Daoud, H., Ezzati-Jivan, N., Dagenais, M.R., 2017. Dynamic trace-based sampling algorithm for memory usage tracking of enterprise applications. In: 2017 IEEE High Performance Extreme Computing Conference. HPEC 2017, IEEE, New York, New York, USA, pp. 1–7. http://dx.doi.org/10.1109/HPEC.2017.8091061.

Ding, R., Zhou, H., Lou, J.G., Zhang, H., Lin, Q., Fu, Q., Zhang, D., Xie, T., 2015. Log2: A cost-aware logging mechanism for performance diagnosis. In: Proceedings of the 2015 USENIX Annual Technical Conference. USENIX ATC 2015, USENIX Association, Santa Clara, CA, USA, pp. 139–150, URL: https://www.usenix.org/conference/atc15/technical-session/presentation/ding.

Dugerdil, P., 2007. Using trace sampling techniques to identify dynamic clusters of classes. In: Proceedings of the 2007 Conference of the Center for Advanced Studies on Collaborative Research. CASCON '07, ACM Press, New York, New York, USA, pp. 306–314. http://dx.doi.org/10.1145/1321211.1321254.

Estabrooks, A., Jo, T., Japkowicz, N., 2004. A multiple resampling method for learning from imbalanced data sets. Comput. Intell. 20 (1), 18–36. http://dx.doi.org/10.1111/j.0824-7935.2004.t01-1-00228.x.

Fei, L., Midkiff, S.P., 2006. Artemis. In: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '06, ACM Press, New York, New York, USA, p. 84. http://dx.doi.org/10.1145/1133981.1133992.

Feng, Y., Dreef, K., Jones, J.A., van Deursen, A., 2018. Hierarchical abstraction of execution traces for program comprehension. In: Proceedings of the 26th Conference on Program Comprehension. ICPC '18, Association for Computing Machinery (ACM), New York, New York, USA, pp. 86–96. http://dx.doi.org/10.1145/3196321.3196343.

Finocchi, I., 2013. Software Streams: Big Data Challenges in Dynamic Program Analysis. In: Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 7921 LNCS, pp. 124–134. http://dx.doi.org/10.1007/978-3-642-39053-1_15.

Gao, L., Lu, M., Li, L., Pan, C., 2017. A survey of software runtime monitoring. In: 2017 8th IEEE International Conference on Software Engineering and Service Science. ICSESS, IEEE, Beijing, China, pp. 308–313. http://dx.doi.org/10.1109/ICSESS.2017.8342921.

Hamou-Lhadj, A., Lethbridge, T.C., 2004. A survey of trace exploration tools and techniques. In: CASCON '04 Proceedings of the 2004 Conference of the Centre for Advanced Studies on Collaborative Research. IBM Press, pp. 42–55, URL: http://dl.acm.org/citation.cfm?id=1034914.1034918.

Hauswirth, M., Chilimbi, T.M., 2004. Low-overhead memory leak detection using adaptive statistical profiling. In: Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems. ASPLOS-XI, ACM Press, New York, New York, USA, pp. 156–164. http://dx.doi.org/10.1145/1024393.1024412.

Horký, V., Kotrč, J., Libič, P., Tůma, P., 2016. Analysis of overhead in dynamic java performance monitoring. In: Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering. ICPE '16, ACM Press, New York, New York, USA, pp. 275–286. http://dx.doi.org/10.1145/2851553.2851569, URL: http://dl.acm.org/citation.cfm?doid=2851553.2851569.

Jung, C., Lee, S., Raman, E., Pande, S., 2014. Automated memory leak detection for production use. In: Proceedings - International Conference on Software Engineering, No. 1. IEEE Computer Society, New York, New York, USA, pp. 825–836. http://dx.doi.org/10.1145/2568225.2568311, URL: http://dl.acm.org/citation.cfm?doid=2568225.2568311.

Kang, P., 2018. Function call interception techniques. Softw. - Pract. Exp. 48 (3), 385–401. http://dx.doi.org/10.1002/spe.2501, URL: http://doi.wiley.com/10.1002/spe.2501.

Las-Casas, P., Mace, J., Guedes, D., Fonseca, R., 2018. Weighted sampling of execution traces. In: Proceedings of the ACM Symposium on Cloud Computing. SoCC '18, ACM Press, New York, New York, USA, pp. 326–332. http://dx.doi.org/10.1145/3267809.3267841, URL: http://dl.acm.org/citation.cfm?doid=3267809.3267841.

Las-Casas, P., Papakerashvili, G., Anand, V., Mace, J., 2019. Sifter: Scalable sampling for distributed traces, without feature engineering. In: Proceedings of the ACM Symposium on Cloud Computing. SoCC '19, Association for Computing Machinery (ACM), New York, New York, USA, pp. 312–324. http://dx.doi.org/10.1145/3357223.3362736, URL: http://dl.acm.org/citation.cfm?doid=3357223.3362736.

Mertz, J., Nunes, I., 2017. Understanding application-level caching in web applications. ACM Comput. Surv. 50 (6), 1–34. http://dx.doi.org/10.1145/3145813, URL: http://dl.acm.org/citation.cfm?doid=3161158.3145813.

Mertz, J., Nunes, I., 2018. Automation of application-level caching in a seamless way. Softw. - Pract. Exp. 48 (6), 1218–1237. http://dx.doi.org/10.1002/spe.2571, URL: http://onlinelibrary.wiley.com/doi/10.1002/spe.2571/abstract.

Mertz, J., Nunes, I., 2019. On the practical feasibility of software monitoring: A framework for low-impact execution tracing. In: Software Engineering for Adaptive and Self-Managing Systems, Vol. 2019-May. SEAMS, IEEE, Montreal, Canada, pp. 169–180. http://dx.doi.org/10.1109/SEAMS.2019.00030.

Mertz, J., Nunes, I., 2021. Tigris: A DSL and framework for monitoring software systems at runtime. J. Syst. Softw. 177, 110963. http://dx.doi.org/10.1016/j.jss.2021.110963.

Miranskyy, A., Hamou-Lhadj, A., Cialini, E., Larsson, A., 2016. Operational-log analysis for big data systems: Challenges and solutions. IEEE Softw. 33 (2), 52–59. http://dx.doi.org/10.1109/MS.2016.33.

Narayanappa, H., Bansal, M.S., Rajan, H., 2010. Property-aware program sampling. In: Proceedings of the 9th ACM SIGPLANSIGSOFT Workshop on Program Analysis for Software Tools and Engineering PASTE 10. ACM Press, New York, New York, USA, p. 45. http://dx.doi.org/10.1145/1806672.1806682, URL: http://portal.acm.org/citation.cfm?doid=1806672.1806682.

Pirzadeh, H., Shanian, S., Hamou-Lhadj, A., Alawneh, L., Shafiee, A., 2013. Stratified sampling of execution traces: Execution phases serving as strata. Sci. Comput. Program. 78 (8), 1099–1118. http://dx.doi.org/10.1016/j.scico.2012.11.002.

Pirzadeh, H., Shanian, S., Hamou-Lhadj, A., Mehrabian, A., 2011. The concept of stratified sampling of execution traces. In: IEEE International Conference on Program Comprehension. IEEE, New York, New York, USA, pp. 225–226. http://dx.doi.org/10.1109/ICPC.2011.17.

Reger, G., Havelund, K., 2016. What Is a Trace? A Runtime Verification Perspective. In: Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 9953 LNCS, Springer, Cham, New York, New York, USA, pp. 339–355. http://dx.doi.org/10.1007/978-3-319-47169-3_25.

Santelices, R., Sinha, S., Harrold, M.J., 2006. Subsumption of program entities for efficient coverage and monitoring. In: Proceedings of the Third International Workshop on Software Quality Assurance. SOQUA 2006, ACM Press, New York, New York, USA, pp. 2–5. http://dx.doi.org/10.1145/1188895.1188899.

Song, L., Lu, S., 2017. Performance diagnosis for inefficient loops. In: Proceedings - 2017 IEEE/ACM 39th International Conference on Software Engineering. ICSE 2017, Institute of Electrical and Electronics Engineers Inc., New York, New York, USA, pp. 370–380. http://dx.doi.org/10.1109/ICSE.2017.41.

Sridharan, M., Fink, S.J., Bodík, R., 2007. Thin slicing. In: ACM SIGPLAN Notices. ACM, New York, New York, USA, pp. 112–122. http://dx.doi.org/10.1145/1273442.1250748.

Toffola, L.D., Pradel, M., Gross, T.R., 2015. Performance problems you can fix: a dynamic analysis of memoization opportunities. In: Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, Vol. 25-30-Oct-. OOPSLA 2015, ACM Press, New York, New York, USA, pp. 607–622. http://dx.doi.org/10.1145/2814270.2814290, http://dl.acm.org/citation.cfm?doid=2814270.2814290.

van Hoorn, A., Waller, J., Hasselbring, W., 2012. Kieker: A framework for application performance monitoring and dynamic software analysis. In: Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering. ICPE '12, Association for Computing Machinery, New York, NY, USA, pp. 247–248. http://dx.doi.org/10.1145/2188286.2188326.

Von Kistowski, J., Herbst, N., Kounev, S., Groenda, H., Stier, C., Lehrig, S., 2017. Modeling and extracting load intensity profiles. ACM Trans. Auton. Adapt. Syst. 11 (4), 1–28. http://dx.doi.org/10.1145/3019596, URL: https://dl.acm.org/doi/abs/10.1145/3019596.

Yuan, E., Esfahani, N., Malek, S., 2014. Automated mining of software component interactions for self-adaptation. In: Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. SEAMS 2014, ACM Press, New York, New York, USA, pp. 27–36. http://dx.doi.org/10.1145/2593929.2593934.

Zavala, E., Franch, X., Marco, J., 2019. Adaptive monitoring: A systematic mapping. Inf. Softw. Technol. 105, 161–189. http://dx.doi.org/10.1016/J.INFSOF.2018.08.013.

Zhou, N., Delaval, G., Robu, B., Rutten, E., Mehaut, J.F., 2016. Autonomic parallelism and thread mapping control on software transactional memory. In: Proceedings - 2016 IEEE International Conference on Autonomic Computing. ICAC 2016, Institute of Electrical and Electronics Engineers Inc., New York, New York, USA, pp. 189–198. http://dx.doi.org/10.1109/ICAC.2016.54.

Jhonny Mertz is a software engineer at HP Inc (Brazil). He obtained his Ph.D. in Computer Science at the Universidade Federal do Rio Grande do Sul (Brazil). His research interests are in the intersection of software engineering and data analytics, particularly interested in leveraging software traces and logs to support software development and operations.

Ingrid Nunes is a product owner at Utimaco (Germany) and Associate Professor at the Institute of Informatics, Universidade Federal do Rio Grande do Sul (Brazil), on leave since January 2022. She obtained her Ph.D. in Informatics at the Pontifical Catholic University of Rio de Janeiro (Brazil), in cooperation with King's College London (UK) and University of Waterloo (Canada). Her main research areas are software design, maintenance and evolution and self-adaptive systems.