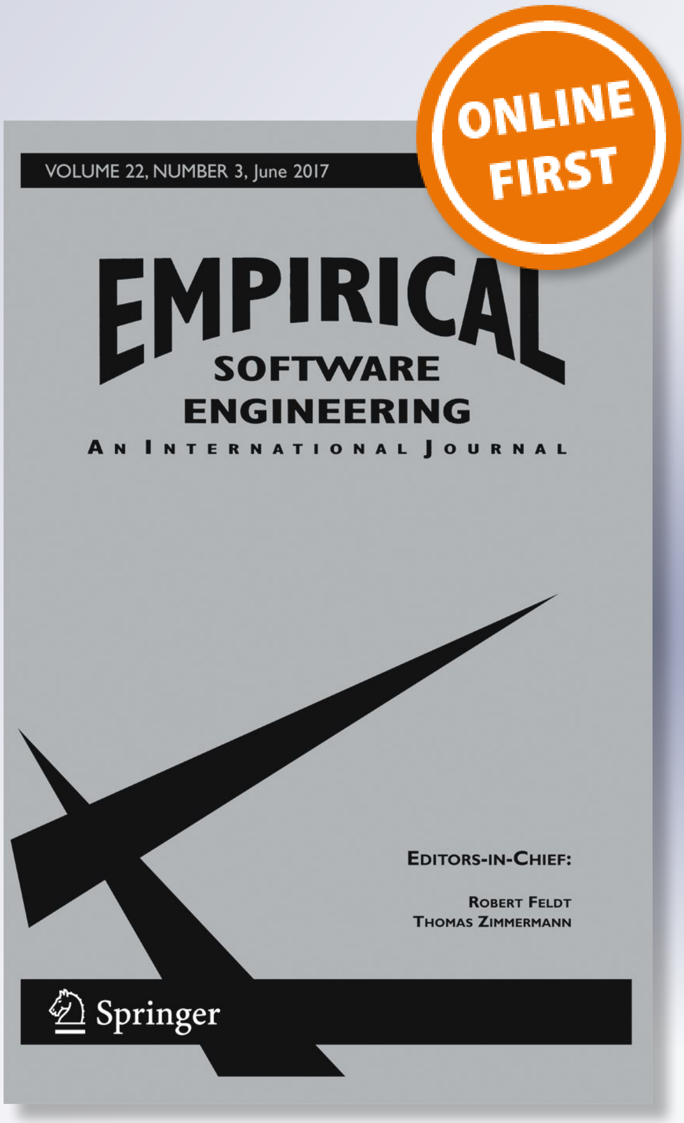
Log4Perf: suggesting and updating logging locations for web-based systems' performance monitoring

Kundi Yao, Guilherme B. de Pádua, Weiyi Shang, Catalin Sporea, Andrei Toma & Sarah Sajedi

Empirical Software Engineering
An International Journal

ISSN 1382-3256

Empir Software Eng
DOI 10.1007/s10664-019-09748-z



Empirical Software Engineering
https://doi.org/10.1007/s10664-019-09748-z

Log4Perf: suggesting and updating logging locations for web-based systems' performance monitoring

Kundi Yao[1] · Guilherme B. de Pádua[1] · Weiyi Shang[1] · Catalin Sporea[2] · Andrei Toma[2] · Sarah Sajedi[2]

Published online: 26 October 2019

Abstract

Performance assurance activities are an essential step in the release cycle of software systems. Logs have become one of the most important sources of information that is used to monitor, understand and improve software performance. However, developers often face the challenge of making logging decisions, i.e., neither logging too little and logging too much is desirable. Although prior research has proposed techniques to assist in logging decisions, those automated logging guidance techniques are rather general, without considering a particular goal, such as monitoring software performance. In this paper, we present Log4Perf, an automated approach that provides suggestions of where to insert logging statement with the goal of monitoring web-based systems' CPU usage. In the first part of our approach, we leverage the performance model's prediction errors to suggest the need for updating logging locations when software evolves. In the second part of our approach, we build and manipulate a statistical performance model to identify the locations in the source code that statistically significantly influence CPU usage. To evaluate Log4Perf, we conduct case studies on two open source systems, i.e., CloudStore and OpenMRS, and one large-scale commercial system. Our evaluation results show that our approach can suggest the need for updating logging locations and identify the logging locations that can be kept unchanged. We manually examine the logging locations that are newly suggested or deprecated. We find that our approach can build well-fit statistical performance models, indicating that such models can be leveraged to investigate the influence of locations in the source code on performance. The suggested logging locations are often in small and simple methods that do not have logging statements, and are not performance hotspots. Our approach can be used to complement traditional approaches that are based on software metrics or performance hotspots. In addition, we identify seven root-causes of these suggested or deprecated logging locations. Log4Perf is integrated into the release engineering process of the commercial software to provide logging suggestions on a regular basis.

Keywords Software logs · Logging suggestion · Performance monitoring · Performance modeling · Performance engineering

Communicated by: Vittorio Cortellessa

✉ Kundi Yao
ku_yao@encs.concordia.ca

Extended author information available on the last page of the article.

1 Introduction

The rise of large-scale software systems, such as web-based systems like Amazon, has imposed an impact on people's daily lives. The increasing importance and complexity of such systems make their quality a critical, yet a hard issue to address. Failures in such systems are more often associated with performance issues, rather than with feature bugs (Weyuker and Vokolos 2000). Therefore, performance assurance activities are an essential step in the release cycle of large software systems.

Monitoring performance of large systems is a crucial task of performance assurance activities. In practice, performance data is often collected either based on system-level information (Cohen et al. 2005), such as CPU usage, or application-level information, such as response time or throughput. In particular, Application Performance Management tools, such as Kieker (Van Hoorn et al. 2012), are widely used in practice. They collect performance data from the systems when they are running in the field environment. However, such system-level or application-level performance data often leads to the challenges of pinpointing the exact location in the source code that is related to performance issues.

On the other hand, the valuable information in logs are widely used in practice to improve the quality of large software systems (Kernighan and Pike 1999; Jiang et al. 2009; Chen et al. 2014; 2016; Shang et al. 2015). Prior research proposed and used logs to monitor and improve software performance (Jiang et al. 2009; Chen et al. 2014; 2016; Shang et al. 2015). The success of those performance assurance techniques depends on the well-maintained logging infrastructure and the high quality of the logs. Although prior research has proposed various approaches to improve the quality of logs (Zhao et al., 2017; Fu et al. 2014; Zhu et al. 2015; Yuan et al. 2011, 2012; Li et al. 2017, 2017), all of these approaches consider logs in general, i.e., not considering the particular need of using logs for performance assurance activities. Therefore, the suggested improvement of logs may not be of interest in performance assurance activities.

Making it worse, software ever evolves while the locations in the source code that contributes to performance modeling also evolve. Intuitively, keeping the update-to-date logging locations requires repetitive performance testing and evaluation in order to understand the performance and the generated logs in the up-to-date version of the systems. However, such a repetitive process not only requires extra effort but may also impact the developer and operators who leverage logs. For example, our prior work requires up to 50 hours of 10 iterations of performance testing to suggest logging locations for a single release of a software system (Yao et al. 2018).

In order to address the above challenges, in this paper, we present an approach that automatically provides logging location suggestion for web-based systems based on the particular interest in performance modeling. Our approach consists of two parts as follows:

Part 1: We leverage statistical analysis to evaluate whether there is a need for updating logging locations, in order to reduce the needed resources for performance testing and evaluation. If the existing logging locations can still model system performance (such as CPU usage) in a new version, we suggest not to update the logging locations. In addition, if there is a need for updating logging locations, our approach evaluates the performance prediction power of every existing logging location to determine whether any of them should be kept in the new version.

Part 2: By knowing that there is a need for updating logging locations, our approach first automatically insert logging statements into the source code. After iteratively conducting performance tests with the system, our approach builds statistical performance models to represent the system performance (such as CPU usage) using logs that are generated by the automatically inserted logging statements in the source code. By improving and analyzing statistical performance models, our approach identifies the logging statements that are statistically significant in explaining the system performance. Such logging statements are suggested to practitioners as potential logging locations for the use of performance assurance activities.

We evaluate our approach with two open source systems, namely OpenMRS and CloudStore, and one commercial system. Our evaluation results[1] show that we can identify the releases without the need for updating logging locations and the logging locations that can be kept unchanged across releases. By focusing on the suggestion of logging locations, we find that we can build statistical performance models with R^2 between 26.9% and 90.2%. By studying the suggested logging locations, we find that they all have a high influence on the CPU usage. These locations cannot be identified using code complexity metrics nor detected as performance hotspots. Finally, we manually identified seven root-causes of newly suggested or deprecated logging locations.

This paper makes the following contributions:

– To the best of our knowledge, our work is the first to provide logging suggestions with the particular goal of performance monitoring.
– We propose a statistically rigorous approach to identifying source code locations that can statistically explain the CPU usage.
– The outcome of our approach can complement the use of traditional code metrics and performance hotspots to assist performance engineers in practice.
– We introduce our approach on suggesting the need for updating logging locations when software evolves.
– We identify seven root-causes of suggesting or deprecating logging locations. Future research can prioritize on these root-causes in logging decision suggestions.

Our previous work (Yao et al. 2018) published in the 9th ACM/SPEC International Conference on Performance Engineering (ICPE 2018) with title "Log4Perf: Suggesting Logging Locations for Web-based Systems' Performance Monitoring" focused on automatically suggesting logging locations in source code where can help monitor the CPU usage (part 2). However, one of the major challenges of adopting our existing work in practice is the resource needed for iterative performance testing and evaluation. In order to address such a challenge, in this paper, we extend our prior work to help practitioners determine when and where to update logging locations for system performance monitoring (part 1).

Our approach is already adopted in an industrial environment and is integrated into a continuous deployment environment. Developers receive logging suggestions from our automated approach regularly to better monitor the system performance in the field.

The rest of this paper is organized as follows: Section 2 presents our automated approach to suggest logging locations. Section 3 and 4 present the case study setup and the results of evaluating our approach by answering five research questions. Section 5 discuss related topics based on the results. Section 6 presents the prior research that is related to this paper. Section 7 presents the threats to the validity of our study. Finally, Section 8 concludes this paper.

[1]The data and setup of our evaluation is shared online: https://github.com/KDYao/Log4Perf_Replication Package.

2 Approach

In this section, we present our approach on suggesting logging locations for software performance monitoring. Our approach consists of two major parts, where each of them aims to address the challenges of identifying logging locations for performance monitoring. In particular, in order to reduce the large amounts of resource needed for performance testing and evolution to suggest logging locations, our approach first applies statistical modeling and analysis to determine whether there is a need for updating logging locations, and which existing logging locations can be kept without changing. Second, once the need for updating logging locations is identified, our approach automatically provides suggestions on logging locations based on iterations of performance testing and modeling. The general overview of our approach is shown in Fig. 1.

Part 1: Identifying the Need for Updating Logging Locations

In this subsection, we present the part 1 of our approach that automatically determines the need for updating logging locations. Intuitively, developers may run performance testing and evaluation with multiple iterations for every version of the software. However, such iterations of performance testing are extremely resource-heavy for development team, leading to the delay of having logging statements in the software. Moreover, the need for updating logging locations does not mean that every existing logging location needs to be discarded. Simply discarding all existing logging locations is impetuous because some logging locations may still potentially influence performance, though they may not perform as significant as they were. Such logging locations should be kept in order to complement the new logging locations from re-applying our logging suggestion approach to the updated software system.

Therefore, this part of our approach 1) checks the effectiveness of the existing logging locations and 2) suggests which logging statements can be kept in the new version in order to minimize the iterations of performance testing.

The overview of this part of the approach is shown in Fig. 2.

Step 1.1: Checking Effectiveness of the Existing Logging Locations

We aim to identify whether the existing logs in the system are already effective in modeling the system performance (like CPU usage). If the model built from existing logging locations cannot provide an accurate prediction, we need to identify new logging locations for the software. We use the prediction errors as a measurement to evaluate the existing logging locations and model on the new version of the system. While we conduct the performance test, we measure the performance counters, i.e., CPU usage, while collecting the logs generated from the existing logging locations. In particular, we choose CPU usage in our approach. There exist other important performance metrics such as memory usage and response time. One may consider using other metrics in this approach.

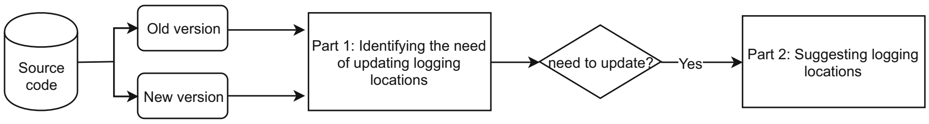
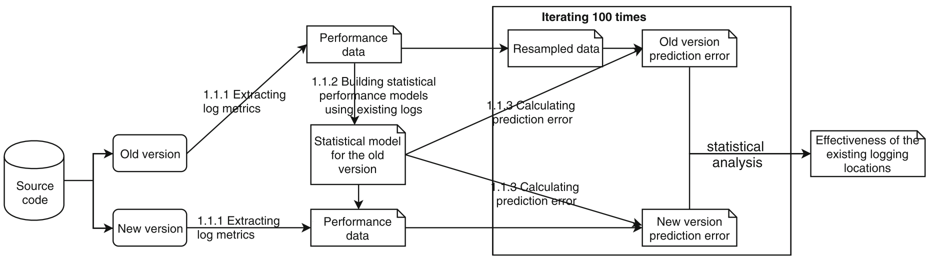
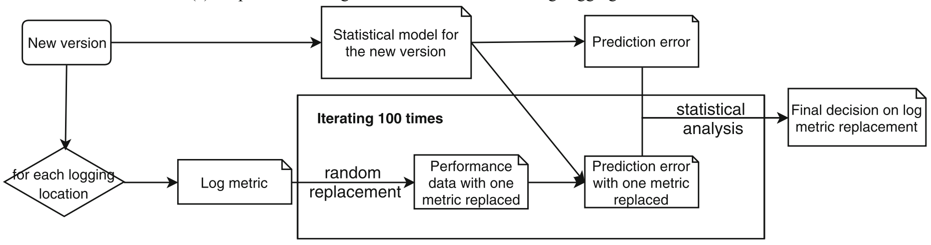

Source code
Old version
New version
Part 1: Identifying the need of updating logging locations
need to update?
Yes
Part 2: Suggesting logging locations

Fig. 1 An overview of our approach


Iterating 100 times
Performance data
Resampled data
Old version prediction error
1.1.1 Extracting log metrics
1.1.2 Building statistical performance models using existing logs
1.1.3 Calculating prediction error
Statistical model for the old version
statistical analysis
Effectiveness of the existing logging locations
Source code
Old version
New version
1.1.1 Extracting log metrics
Performance data
1.1.3 Calculating prediction error
New version prediction error

(a) Step 1.1: checking effectiveness of the existing logging locations


New version
Statistical model for the new version
Prediction error
Iterating 100 times
statistical analysis
Final decision on log metric replacement
for each logging location
Log metric
random replacement
Performance data with one metric replaced
Prediction error with one metric replaced

(b) Step 1.2: identifying the logging statement that can be kept in the new version

Fig. 2 An overview of the part 1 of our approach that determines the need for updating logging locations

1.1.1 *Extracting Log Metrics*

We run performance tests for our subject systems and monitor their CPU usage during the test. After the performance test, we parse the generated logs (including both web logs and execution logs) that are already generated during the execution of the system. In particular, we keep the time stamp of the logs and the event of the logs (e.g., a RESTFul web request).

We then calculate log metrics based on those logs. Each value of each log metric L is the number of times that each logging location or web request executes during the period. For example, if a log event *user login* is executed 10 times during a 30-second time period, the metric *user login*'s value is 10 for that period. Table 1 shows an illustrative example piece of data for 10 time periods (300 seconds) from two logs: one web log (*index.jsp*) and web execution log (*user login*).

1.1.2 *Building Statistical Performance Models using Existing Logs*

We follow a model building approach that is similar to the approach from prior software performance research (Shang et al. 2015; Cohen et al. 2005; Xiong et al. 2013). We build a linear regression model (Freedman 2009) to model the CPU usage of the software. We choose linear regression model because: 1) the goal of the approach is not to build a perfect model but to interpret the model easily instead, and 2) prior research used such modeling techniques to model software performance (Cohen et al. 2005; Xiong et al. 2013; Farshchi et al. 2015).

We use the log metrics that are generated from existing logs as independent variables. All these independent variables are used to build a model and explain the changes of the dependent variable. The linear regression model is built using the *glm()* function in R. The dependent variable of the model is the performance metrics that are collected during applying the load on the software system, i.e., CPU usage.

Table 1 An illustrative example of log metrics and performance metric data from the old version

	Log metrics		Performance metric
Time periods	user login	index.jsp	CPU
1 second – 30 second	30	14	146.39
31 second – 60 second	36	21	187.17
61 second – 90 second	20	15	137.68
91 second – 120 second	19	17	134.03
121 second – 150 second	26	22	165.16
151 second – 180 second	17	19	143.21
181 second – 210 second	13	14	126.35
211 second – 240 second	27	24	157.5
241 second – 270 second	24	13	139.81
271 second – 300 second	26	16	151.93

The formula of the performance model built from our illustrative example will be as follows:

$$CPU = \beta + \alpha_1 \times user\_login + \alpha_2 \times index.jsp \tag{1}$$

where β is the intercept and α_1 and α_2 are the coefficients. By building the model, we can obtain a formula as follows:

$$CPU = 71.2 + 1.83 \times user\_login + 1.953 \times index.jsp \tag{2}$$

1.1.3 Calculating Prediction Error

We use the built performance model from the last step and the logs to predict the CPU usage. We compare the predicted value and the actual value of the system and calculate prediction error as $\frac{|predictedCPU - actualCPU|}{actualCPU}$. The distribution of the prediction error of the new version represents how well can the existing logging locations model the CPU usage of the new version of the system. Table 2 shows an illustrative example of log metrics and CPU usage data from the new version of the system. The table also shows the predicted CPU value from the model that is built from step 1.1.2 and the corresponding prediction error.

Intuitively, one may use a threshold (e.g., 5%) to determine whether the prediction error is too high, i.e., the need for updating the logging locations. However, such thresholds may vary between systems, releases, or even different workloads. In addition, the choice of thresholds is typically based on experience or gut feeling. Therefore, we use the old version to calculate prediction error using its performance testing data as a baseline.

It is obvious that directly using both the model and data from the old version (although from different runs) would have a lower prediction error than using the model from the old version and data from the new version. In order to avoid such bias, we use *bootstrap* (Bootstrap 2017) to generate a new dataset from the old version to calculate the distribution of prediction error from the old version. Bootstrap will randomly resample existing data with replacement in order to reduce the bias. By using bootstrap we can derive more robust and statistical rigorous result as a baseline for comparison. We resample the data in the illustrative example and the data after resampling is shown in Table 3.

Table 2 An illustrative example of log metrics and performance metric data from the new version and prediction error

	Log metrics		Performance metric	Predicted performance metric	
Time periods	user login	index.jsp	CPU	CPU	Prediciton error
1 second – 30 second	25	15	135.15	146.24	8.2%
31 second – 60 second	21	25	166.03	158.45	0.4%
61 second – 90 second	41	31	188.61	206.76	9.6%
91 second – 120 second	30	17	148.99	159.29	6.9%
121 second – 150 second	26	12	131.97	142.21	7.8%
151 second – 180 second	29	21	152.5	165.27	8.4%
181 second – 210 second	33	24	158.01	178.45	13.0%
211 second – 240 second	41	19	172.79	183.32	6.1%
241 second – 270 second	35	16	149.29	166.49	11.5%
271 second – 300 second	21	22	144.53	152.59	5.6%

Finally, with the two distribution of prediction errors at hand, we compare the two distributions similar to previous studies (Chen et al. 2016a), using Wilcoxon rank sum test (Moore et al. 2012) and Cliff's d (1993). In particular, Wilcoxon rank sum test is used to compare the mean values of prediction errors and determine whether two distributions of prediction errors are significantly different from each other. If our predictions vary a lot from each other, our suggested logging locations may not be a suitable option for an upgraded system. We consider the results reside in 95% confidential interval (p-value $\leqslant 0.05$) as indicators of statistically significant difference. However, even if two datasets are significantly different, the actual effect may be negligible. Therefore, we also calculate the actual effect size using Cliff's d, to further illustrate the impact size on predicted CPU usage changes.

Table 3 Log metrics and performance metric data after resample from the illustrative example in Table 1 and prediction

	Log metrics		Performance metric	Predicted performance metric	
Time periods	user login	index.jsp	CPU	CPU	Prediction error
1 second – 30 second	36	21	187.17	178.08	4.9%
31 second – 60 second	13	14	126.35	122.33	3.2%
61 second – 90 second	13	14	126.35	122.33	3.2%
91 second – 120 second	13	14	126.35	122.33	3.2%
121 second – 150 second	24	13	139.81	140.51	0.5%
151 second – 180 second	13	14	126.35	122.33	3.2%
181 second – 210 second	30	14	146.39	153.43	4.8%
211 second – 240 second	20	15	137.68	137.09	0.4%
241 second – 270 second	13	14	126.35	122.33	3.2%
271 second – 300 second	17	19	143.21	139.41	2.7%

Cliff's d is widely used in previous studies to quantify difference between two datasets. The threshold of Cliff's d is defined as:

$$\textit{effect size} = \begin{cases} negligible & \text{if Cliff's d} \leqslant 0.147 \\ small & \text{if } 0.147 < \text{Cliff's d} \leqslant 0.33 \\ medium & \text{if } 0.33 < \text{Cliff's d} \leqslant 0.474 \\ large & \text{if Cliff's d} > 0.474 \end{cases}$$

In our illustrative example, by comparing the distribution of prediction errors in the old version and the new version (the prediction error column in Tables 2 and 3), the p-value from the Wilcoxon rank sum test is 0.0003 and Cliff's d is 0.96 (large effect size). From the result, we can tell that there exists a significant difference between the prediction errors, and the magnitude of such difference is large. This result points out that the previous prediction model is not capable of explaining the CPU usage changes accurately after the system update, the replacement of deprecated prediction model is needed.

We repeat this process (starting from resampling the old version's data with bootstrap) for 100 times. For each time, we provide results of whether the two sets of distributions are statistically significant and its effect size. We report the results to developers in order to determine the need for updating logging locations.

Step 1.2: Identifying the Logging Statement that can be Kept in the New Version

In this step, we identify the logging statements that still can help model CPU usage in the current model and the ones that cannot. In general, we replace the values of each log metric by random values and evaluate the impact on model prediction error. If the prediction error is significantly increased, it means this metric still helps model CPU usage. Hence, the logging location of the metric should be kept. On the other hand, if the prediction error is not impacted significantly by replacing the metric, the logging location of the metric would not help model CPU usage. Therefore, the logging location should be removed.

In particular, for each metric in the model, we replace its values by randomly selecting a value from the range between its minimum and maximum values in the original data. After replacing these values, we re-calculate the distribution of prediction errors. We compare the distribution of prediction errors with and without replacing values of that metric using Wilcoxon rank sum test and Cliff's d. Then we evaluate whether to keep the metric based on whether the difference is statistically significant (p-value $\leqslant$ 0.05) and its effect sizes.

In our illustrative example, we replace the log metric *index.jsp* from Table 2 by a random variable and recalculate the prediction error as shown in Table 4. From our illustrative example, the p-value from the Wilcoxon rank sum test is 0.8534 and Cliff's d is 0.06 (negligible effect). This result indicates that the CPU usage does not have a significant and large difference, even when we replace the value of a metric by random values. In this case, the corresponding log metric does not contribute to explaining the CPU usage anymore, so such metric should be replaced. In order to reduce the bias from our metric value random replacement, we repeat this process by 100 times, similar to step 1.1.

Part 2: Suggesting Logging Locations

From the last part of our approach, we know whether there is a need for updating logging locations for performance monitoring and whether any existing logging locations can be kept. With such knowledge, in this subsection, we present the part 2 of our approach that automatically suggests logging locations for software performance monitoring. To reduce the performance overhead caused by introducing instrumentation into the source code, we first leverage the readily available web logs to build a statistical performance

Table 4 Log metrics and performance metric data after replacing the *index.jsp* metric from the illustrative example in Table 2 to random values and prediction error

	Log metrics		Performance metric	Predicted performance metric	
Time periods	user login	index.jsp	CPU	CPU	Prediciton error
1 second − 30 second	25	26	135.15	167.72	24.1%
31 second − 60 second	21	28	166.03	164.30	1.0%
61 second − 90 second	41	28	188.61	200.90	6.5%
91 second − 120 second	30	22	148.99	169.06	13.4%
121 second − 150 second	26	28	131.97	173.45	31.4%
151 second − 180 second	29	15	152.50	153.56	0.7%
181 second − 210 second	33	15	158.01	160.88	1.8%
211 second − 240 second	41	26	172.79	197.00	14.0%
241 second − 270 second	35	29	149.29	191.87	28.5%
271 second − 300 second	21	19	144.53	146.73	1.5%

model, and we identify the web requests that are statistically significantly performance-influencing. In the second step, we only focus on the methods that are associated with the performance-influencing web requests and identify which method is statistically significantly performance-influencing. Finally, we focus on the basic blocks in the source code that are associated with the performance-influencing methods, and we identify and suggest the code blocks where logging statements should be inserted.

For each step, we apply a workload on the subject system while monitoring its performance. Afterward, we build a statistical model for the performance of the subject system using the readily available web logs and the automatically generated logs from instrumentation during the workload. Using the statistical performance model, we identify the statistically significant performance-influencing logging statements.

The overview of the part 2 of our approach is shown in Fig. 3.

Step 2.1: Identifying Performance-Influencing Web Requests

In the first step, we aim to identify the source code associated with web requests that influence system performance.

2.1.1 Parsing Web Logs

Similar to step 1.1.1, we parse the generated web logs. We then calculate log metrics based on the web logs. In order to illustrate our approach, we show an illustrative example of log metrics with the two web requests *index.jsp* and *purchase.jsp* in Table 5.

2.1.2 Building Statistical Performance Models using Web Logs

We first follow a model building approach that is similar to step 1.1.2. After building a linear regression model for the CPU usage of the software, we examine each independent variable, i.e., log metric, to see how statistically significant it is to the model's output, i.e., CPU usage. In particular, we only consider the log metrics that have p-value ≤ 0.05. Since each log metric represents the number of times that the associated source code of each web request executes, the significance of a log metric shows whether the execution of the web

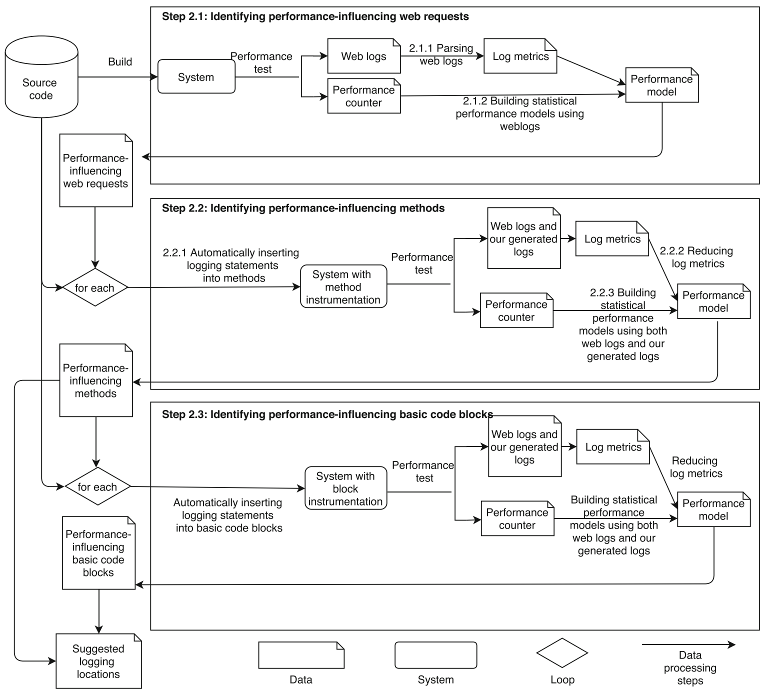

Step 2.1: Identifying performance-influencing web requests
Source code
Build
System
Performance test
Web logs
2.1.1 Parsing web logs
Log metrics
Performance counter
2.1.2 Building statistical performance models using weblogs
Performance model
Performance-influencing web requests
Step 2.2: Identifying performance-influencing methods
2.2.1 Automatically inserting logging statements into methods
for each
System with method instrumentation
Performance test
Web logs and our generated logs
Log metrics
2.2.2 Reducing log metrics
Performance counter
2.2.3 Building statistical performance models using both web logs and our generated logs
Performance model
Performance-influencing methods
Step 2.3: Identifying performance-influencing basic code blocks
for each
Automatically inserting logging statements into basic code blocks
System with block instrumentation
Performance test
Web logs and our generated logs
Log metrics
Reducing log metrics
Performance counter
Building statistical performance models using both web logs and our generated logs
Performance model
Performance-influencing basic code blocks
Suggested logging locations
Data
System
Loop
Data processing steps

Fig. 3 An overview of the part 2 of our approach that suggests logging locations for performance monitoring

log associated source code has a statistically significant influence on the software CPU usage. Based on the list of statistically significant log metrics, we identify the performance-influencing web requests.

From our illustrative example shown in Table 5, we build a linear regression model where only *purchase.jsp* is significant in the model while *index.jsp* is not. Therefore, we only keep *purchase.jsp* in the model.

Table 5 An illustrative example of log metrics with two web requests and performance metric data

	Log metrics		Performance metric
Time periods	index.jsp	purchase.jsp	CPU
1 second − 30 second	33	78	170.01
31 second − 60 second	15	53	112.42
61 second − 90 second	53	62	138.09
91 second − 120 second	32	59	119.74
121 second − 150 second	45	68	142.03
151 second − 180 second	31	61	126.82
181 second − 210 second	29	49	110.10
211 second − 240 second	24	55	116.28
241 second − 270 second	19	58	127.21
271 second − 300 second	36	54	119.67

Table 6 An illustrative example of log metrics with one web request and two associated methods and performance metric data

	Log metrics			Performance metric
Time periods	index.jsp	check_inventory	make_payment	CPU
1 second – 30 second	48	48	26	168.02
31 second – 60 second	42	41	22	156.47
61 second – 90 second	29	37	17	126.10
91 second – 120 second	45	49	24	163.77
121 second – 150 second	33	51	26	147.41
151 second – 180 second	41	63	33	177.98
181 second – 210 second	39	42	21	158.92
211 second – 240 second	36	58	27	149.21
241 second – 270 second	25	55	27	138.03
271 second – 300 second	42	46	24	168.25

Step 2.2: Identifying Performance-Influencing Methods

In the second step, we focus only on the performance-influencing web requests, and we aim to identify which methods in the source code are statistically significantly influencing performance (CPU usage). To reduce the performance overhead of the instrumentation, we note that every time we only focus on *one* performance-influencing web request. If multiple web requests are found performance-influencing, we repeat this step for every one of them.

2.2.1 Automatically Inserting Logging Statements into Methods

In this step, we automatically insert a logging statement into every method that is associated with the performance-influencing web requests. We use source code analysis frameworks, such as Eclipse JDT (2017) and .NET Compiler Platform ("Roslyn") (2017), to parse the source code and to identify the associated methods in the source code. We automatically insert a logging statement based on *Log4j2* [2] or *Log4Net* [3] and *Log4Net.Async* [4] at the beginning of each method source code. Since the goal of our approach is only suggesting the location to insert logging statement, we only print the time stamp and the method signature using the logging statement. After re-building the systems and applying performance tests to each subject system, logs will be generated automatically.

Similar to step 2.1, we parse both the web logs and the logs that are generated by our inserted logging statement. Then we generate log metrics based on these logs.

In our illustrative example, we may identify that *purchase.jsp* calls two methods: *check_inventory* and *make_payment*. We automatically insert a logging statement into each method. The new illustrative data of at the method level is shown in Table 6.

2.2.2 Reducing Metrics

Intuitively, methods that never execute during a workload do not influence the CPU usage of the system. Although the methods with a constant number of execution times could have

[2] https://logging.apache.org/log4j/2.x/

[3] https://logging.apache.org/log4net/

[4] https://github.com/cjbhaines/Log4Net.Async/

performance influence, such impact cannot be quantified by our analysis, since we cannot measure the CPU usage variance that is introduced by such method. Hence, we first remove any log metric that has constant values in the dataset. Methods may often be called together, or one method may always call another one. In such cases, not all methods need to be logged. Hence, we perform a correlation analysis on the log metrics (Kuhn 2008). We used the Pearson correlation coefficient among all metrics from one environment. We find the pair of log metrics that have a correlation value higher than 0.9. From these two log metrics, we remove the metric that has a higher average correlation with all other metrics. For example, among metrics A, B and C, if metrics A and B have the highest correlation 0.95, we need to remove one metric between them from the dataset. If we find the correlation between A and C (e.g., 0.7) is higher than the correlation between B and C (e.g., 0.6), we will remove metric A, which has a higher average correlation with other metrics, from our dataset. We repeat this step until there exists no correlation higher than 0.9.

We then perform redundancy analysis on the log metrics. The redundancy analysis would consider a log metric redundant if it can be predicted from a combination of other metrics (Harrell 2001). We use each log metric as a dependent variable and use the rest of the log metrics as independent variables to build multiple regression models. We calculate the R^2 of each model and if the R^2 is larger than a threshold (0.9), the current dependent variable (i.e., log metric) is considered redundant. We then remove the metric with the highest R^2 and repeat the process until no log metric can be predicted with R^2 higher than the threshold. For example, if method *foo* can be linearly modeled by the rest of the metrics with $R^2 > 0.9$, we remove the metric for method *foo*.

By performing this step on our illustrative data in Table 6, we only keep two log metrics *index.jsp* and *check_inventory*.

2.2.3 Building Statistical Performance Models using Both Web Logs and our Generated Logs

In this step, we build a similar statistical model as step 2.2. As a difference, we do not include the log metrics from web logs that are found not performance influencing from step 2.2. We follow the same model building process and the same way of identifying statistically significant log metrics. The outcome of this step is the methods that are statistically significantly performance-influencing. After building the linear regression model using our illustrative data in Table 6 with only two log metrics *index.jsp* and *check_inventory*, we can obtain a formula as follows:

$$CPU = 48.56 + 1.84 \times index.jsp + 0.75 \times check\_inventory \tag{3}$$

Step 2.3: Identifying Performance-Influencing Basic Code Blocks

A method may be long and may consist of many basic blocks. It may be the case that only a small number of basic blocks are performance-influencing. Therefore, in the final step, we focus only on the performance-influencing methods, and we aim to identify which basic code block is performance-influencing. Similarly, every time we only focus on one method. If multiple methods are found performance-influencing, we repeat this step for each method.

We use the code analysis frameworks to identify basic blocks of each performance-influencing methods. If a performance-influencing method only contains one basic block, i.e., there is no inner block that exists in that specific method (e.g., getter and setter methods), we do not proceed with this step. For the methods with multiple basic blocks, similar to step 2.2, we automatically insert logging statement into every basic block and generate log metrics by both the web logs and our generated logs. We also follow a similar approach as step 2.2 to identify which code block is statistically significantly influencing performance (CPU usage). We then automatically suggest inserting logging statements into the corresponding basic code blocks. If none of the log metrics from basic blocks are significant, we suggest developers insert logging statement at the beginning of the method itself.

From our illustrative example, function *check_inventory* contains two basic blocks (e.g., for block and if block) inside the method body. In order to track the execution path within the method, we insert logging statements into each block recursively.

```
function int check_inventory(item){
   products <- product list in storage
   for each p in products do
      log.info("check_inventory:4")
      if (p.id == item.id) then
         log.info("check_inventory:6")
         return getProductCount(p.id)
      end if
   end for
end function
```

After inserting logging statements into the block level, the data in our illustrative example is shown in Table 7. We use the data to build a performance model. By examining the model, only two log metrics *index.jsp* and *check_inventory:6* are statistically significant. Therefore, we only keep the logging locations that correspond to these two metrics, and the formula of the final performance model is as follows:

$$CPU = 40.52 + 2.42 \times check\_inventory : 6 \tag{4}$$

Table 7 An illustrative example of log metrics with one web request, two block-level logs and performance metric data

	Log metrics			Performance metric
Time periods	index.jsp	check_inventory:4	check_inventory:6	CPU
1 second – 30 second	39	851	45	144.67
31 second – 60 second	51	824	37	141.80
61 second – 90 second	25	903	34	104.97
91 second – 120 second	56	741	42	159.13
121 second – 150 second	44	698	48	157.26
151 second – 180 second	48	598	45	156.16
181 second – 210 second	35	642	47	144.50
211 second – 240 second	42	796	39	135.40
241 second – 270 second	47	674	41	145.82
271 second – 300 second	36	563	46	142.87

Table 8 Overview of of the releases of our subject systems to evaluate part 2

Subjects	Version	SLOC (K)	# files	# methods
CloudStore	v2	7.7	98	995
OpenMRS	2.0.5	67.3	772	8,361
ES	2017	$> 2,000$	$> 9,000$	$> 100,000$

3 Case Study Setup

In this section, we present the setup of our case study, including the subject systems, the workload, and the experimental environment.

3.1 Subject Systems and their Workloads

We evaluate our approach with open-source software, including *OpenMRS* and *CloudStore*, and one commercial software, *ES*. Since we need to consider the evolution of each subject systems, we identify different releases of each subject system. During our study, we find that CloudStore only has two releases and there only exist minor (like documentation) changes between the two releases. Therefore, we only consider one release from CloudStore in this study. In addition, we find that OpenMRS 2.0 and 2.1 consist parallel development. Hence, we suggest logging locations for OpenMRS 2.0 and 2.1 separately. The details of each subject system are shown in Tables 8 and 9.

OpenMRS is an open-source patient-based medical record system commonly used in developing countries. *OpenMRS* is built by an open community that aims to improve health-care delivery through a robust, scalable, user-driven, open source medical record system platform. Their application design is customizable with low programming requirements, using a core application with extendable modules. We choose *OpenMRS* since it is highly concerned with scalability and its performance has been studied in prior research (Chen et al. 2016b). *OpenMRS* provides a web-based interface and RESTFul services. We deployed the *OpenMRS* version 2.0.5 and the data used are from MySQL backup files that are provided by *OpenMRS* developers. The backup file contains data for over 5K patients and 476K observations. We use the RESTFul API test cases created by Chen et al. (2016b). The test simulates 30 users. The tests are composed of various searches, such as: by patient, concept, encounter, and observation, and editing/adding/retrieving patient information. The tests include randomness to simulate real-world workloads better. We randomly choose how

Table 9 Overview of each release of the subject systems to evaluate part 1

Project	Version compare	# Source lines added	# Source lines deleted	# Source file changed
OpenMRS	2.0.5 to 2.0.6	544	83	29
	2.1.0 to 2.1.1	499	64	24
	2.1.1 to 2.1.2	273	74	21
	2.1.2 to 2.1.3	261	31	9
ES	2018Jan to 2018Mar	> 30K	> 20K	> 300
	2018Mar to 2018Apr	> 30K	> 20K	> 500
	2018Apr to 2018Jun	> 20K	> 10K	> 200

many times each user send each web request. In addition, the choice of sending which web request is randomized, and the input content of each web request is randomized as well. We keep the workload running for five hours. To minimize the noise from the system warmup and cool-down periods, we do not include the data from the first and last half an hour of running the workload. In the end, we keep four hours of data from each performance test.

CloudStore is an open-source sample e-commerce web application developed to be used for the analysis of cloud characteristics of systems, such as capacity, scalability, elasticity, and efficiency. It follows the functional requirements defined by the TPC-W standard for verifiable transaction processing and database benchmarks data (TPC-W 2017). It was developed to validate the European Union funded project called CloudScale (2017). We choose *CloudStore* due to its importance in improving cloud systems performance and scalability. It has also been studied in prior research (Chen et al. 2016b). We deployed the *CloudStore* version v2 and the data used was generated using scripts provided by *CloudStore* developers. The generated data for CloudStore contains about 864K customers, 777K orders, and 300 items. We use the test cases created by Chen et al. (2016b) to cover searching, browsing, adding items to shopping carts, and checking out. The test simulates 150 users and we run the performance tests with the same length as OpenMRS.

ES is a commercial software that provides government-regulation related reporting services. The service is widely used as the market leader of its domain. Compare to the other two systems, *ES* is a larger-scale system, which serves customers worldwide and is currently undergoing active maintenance and development activities on a daily basis. Because of a non-disclosure agreement, we cannot reveal additional details about the system. We do note that it has over ten years of history with more than two million lines of code that are based on Microsoft .Net. We run a typical loading testing suite as the workload of the system.

3.2 Experimental Environment

The experimental environment for the open-source software is set up on three separate machines. The first machine is the database server; the second is the web server in which the web application was deployed and, finally, the third machine simulates users using the JMeter load driver (Apache JMeter 2015). These machines have the same hardware configuration, which is 8 G of RAM and Intel Core i5-4690 @ 3.5 GHz quad-core CPU. They all run the Linux operating system and are connected to a local network.

We use *PSUtil* (2017) to monitor the CPU usage of the software. To minimize the noise of other background processes, we only monitor the process of the subject system that is under the workload. We monitor the CPU usage during the workload for every 10 seconds. In particular, similar to prior research (Syer et al. 2017; Alghmadi et al. 2016), CPU percentage of the monitored process between two timestamps are calculated as the CPU usage of the corresponding workload during the period. In our case study, we only consider CPU usage while one may consider using other metrics such as memory and response time to complement the case study of our approach.

The experimental environment for *ES* is an internal dedicated performance testing environment, also with three machines. The testing environment is deployed with performance monitoring infrastructure. Similar to the open-source software, we monitor the CPU usage of the process of *ES* for every 10-seconds and use a logging library to generate automatically instrumented logs.

To combine the two datasets of performance metrics (CPU usage) and logs, and to further reduce the impact of recording noises, we calculate the mean values of the performance metrics in every 30 seconds. Then, we combine the datasets of performance metrics and system throughput based on the time stamp on a per 30-seconds basis. A similar approach has been applied to address mining performance metrics challenges (Foo et al. 2010). We use *Log4J2*'s asynchronous logging to generate the automatically instrumented logs since it is shown to have the smallest performance overhead (Log4J Async 2017).

4 Case Study Results

In this section, we present our case study results by answering five research questions. For each research question, we present the motivation of asking the question, our approach to answer it and its corresponding results.

4.1 RQ1: How Often do Logging Locations Need to be Updated?

Motivation Source code changes are inevitable during the software development process. It is not realistic to expect logging locations to perform as significant indicators to explain the variance of performance variances at all times. Performance influential logging locations should also be updated as source code evolves.

The *goal* of this RQ is to evaluate if our proposed approach can determine the need for updating logging locations.

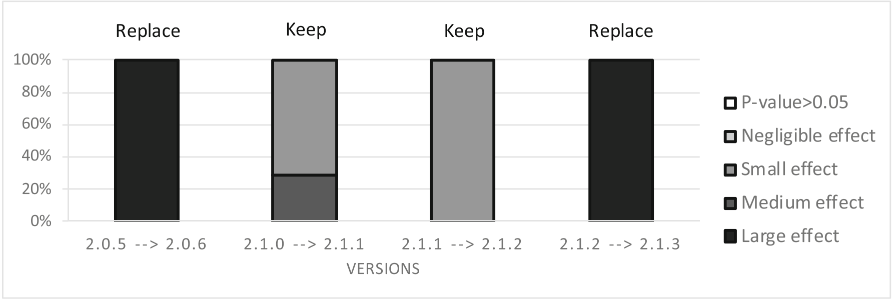
Approach We apply our approach to release 2.0.5, 2.0.6, 2.1.0 and 2.1.3 of OpenMRS. We also apply our approach to the four releases from ES. For each release, we make our decision of whether to update logging locations based on the distribution of model prediction errors (see Section 2, the distribution is generated from 100 times bootstrap resampling to evaluate prediction error). In addition, for the releases that we decide to update logging locations, we apply our approach to select the logging locations to be kept. The *metric* of this RQ is the number of subject releases that needs an update to their logging locations.

Results We use our approach to identify the need for updating logging locations. Tables 10 and 11 show our results in the need for updating logging locations. The prediction errors between every two releases are repeated 100 times for every release. If the majority results indicate that there exists significant difference with medium or large effect sizes between two releases, we consider that the model from the old release is not applicable to explain the CPU usage in the new release, i.e., there is a need for updating logging locations for better performance modeling. For example, in Table 10, number 100 in the large effect size row means that, in the 100 times of bootstrapping, for all 100 times, there exists a statistically significant difference between the prediction error in the old release and in the new release. Therefore, we decide to replace the model. The number 29 at the row of medium effect size and the number 71 at the row of small effect size row imply that out of the 100 times bootstrapping, only 29 times there exists statistically significant differences with medium effect sizes and in 71 times, there exist only small effect sizes. Hence, we conservatively keep the model.

In three releases in our experiment, we make the decision of updating logging locations. In these releases, all prediction errors have statistically significant differences between old and new releases with large effects sizes. On the other hand, for the releases that we decide

Table 10 Decisions of the need for updating logging locations for OpenMRS

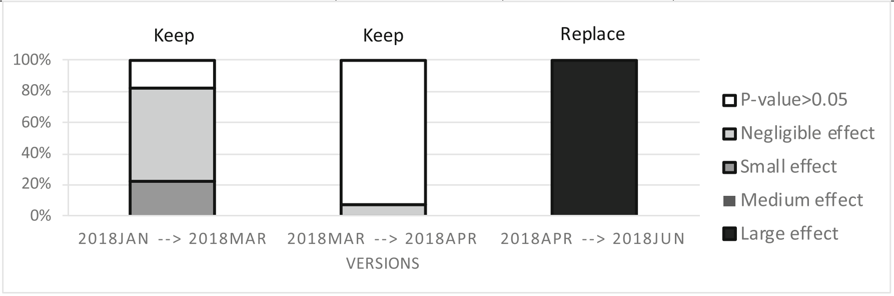
Version		2.0.5 ->2.0.6	2.1.0 ->2.1.1	2.1.1 ->2.1.2	2.1.2 ->2.1.3
# p-value >0.05		0	0	0	0
# Effect size when p-value <0.05	Large	100	0	0	100
	Medium	0	29	0	0
	Small	0	71	100	0
	Negligible	0	0	0	0
Decision		Replace	Keep	Keep	Replace



to keep the logging locations, the majority of the prediction errors have insignificant differences, or with small or negligible effect sizes. The clear difference between the results when we decide to keep and to update logging locations indicates that our approach can be easily adopted by practitioners without a need for tuning thresholds. By comparing the results in Table 9, we find that between those releases where logging locations changes are suggested, there does not always exist larger sizes of code churn. Therefore, simply deciding whether to update logging statements from the magnitude of source code changes is untenable.

Table 11 Decisions of the need for updating logging locations for ES

Version		2018Jan –>2018Mar	2018Mar –>2018Apr	2018Apr –>2018Jun
# p-value >0.05		18	93	0
# Effect size when p-value <0.05	Large	0	0	100
	Medium	0	0	0
	Small	22	0	0
	Negligible	60	7	0
Decision		Keep	Keep	Replace



Most of the old logging locations are suggested to be discarded if there is a need for update logging locations. We choose versions where we decide to update logging locations from Tables 10 and 11. Based on the result from step 1.1, we only focus on the releases where performance model updates are needed, to evaluate the necessity of replacing each logging location. Similarly, the process is also repeated 100 times. If the majority of the comparison results indicate the significant difference and non-negligible effect size, this implies that such metric has sufficient power to explain the model, thus we decide to keep the corresponding log metric. For example, in Table 12, both logging locations at *ConceptServiceImpl.300* and *ConceptServiceImpl.929* have all 100 times with p-value smaller than 0.05, i.e., replacing the metrics of each logging location with random variables would not impact the model. Therefore, both logging locations should be discarded.

As we can see in Tables 12 and 13, most of the previously suggested logging locations should be discarded after software system updates. According to our result, replacing most of the existing logging locations with random data does not produce a statistically significant impact on the model. Therefore, most of the existing logging locations would not influence CPU usage in the new version. The only outstanding logging location is *BaseOpenmrsObject.81* in OpenMRS when updating from release 2.1.2 to 2.1.3. Replacing this logging location with random data would introduce statistically significantly different prediction errors, while the effect sizes are small in most of the iterations. We first made a decision to discard this logging location. However, when applying our approach to suggest logging locations for OpenMRS 2.1.3, this location is suggested by our approach again. This result shows that even the small significant impact on the model is important.

We use our approach to suggest the need for updating logging locations in a new version of the system. If it is decided to update logging locations, most of the existing logging locations need to be discarded.

Table 12 Decisions of discarding or keeping each logging locations in OpenMRS

Version		2.0.5 –>2.0.6		2.1.2 –>2.1.3
Logging locations		ConceptServiceImpl.300	ConceptServiceImpl.929	BaseOpenmrsObject.81
# p-value >0.05		100	100	0
#Effect Size when p-value <0.05	Large	0	0	0
	Medium	0	0	0
	Small	0	0	86
	Negligible	0	0	14
Decision		Discard	Discard	Keep

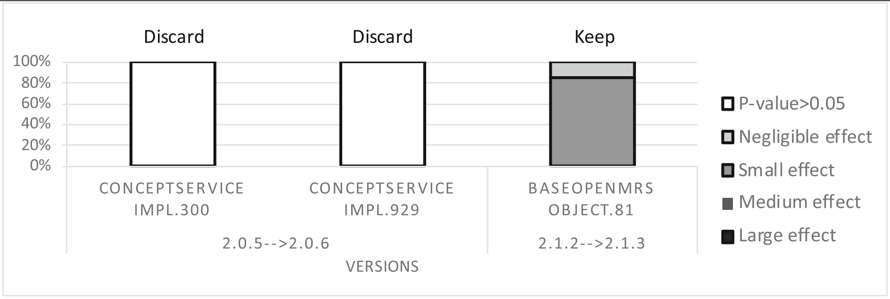
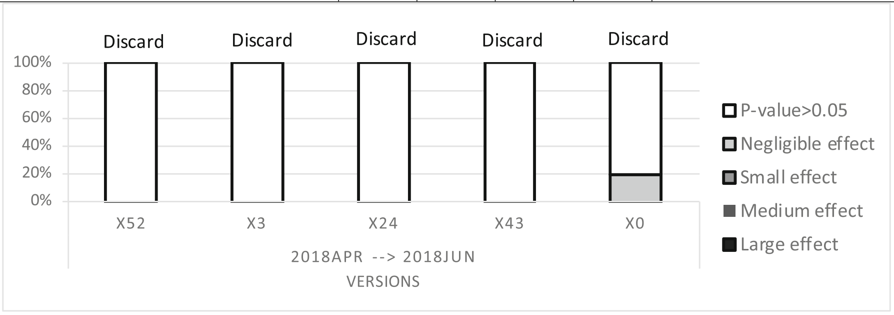


Table 13 Decisions of discarding or keeping each logging locations in ES

Version		2018Apr –>2018Jun				
Logging locations		X52	X3	X24	X43	X0
# p-value >0.05		100	100	100	100	80
# Effect Size when p-value <0.05	Large	0	0	0	0	0
	Medium	0	0	0	0	0
	Small	0	0	0	0	0
	Negligible	0	0	0	0	20
Decision		Discard	Discard	Discard	Discard	Discard



4.2 RQ2: How Well can we Model System Performance (CPU Usage)?

Motivation The success of our approach depends on the ability to build well fit statistical models for CPU usage. The *goal* of this RQ is to assess the ability to build statistical models for CPU usage. If the models built by our approach are of low quality, we cannot use such models to understand the influence of logged source code locations (i.e., log metrics) to the CPU usage. Additionally, the automatically inserted logging statements have an impact on CPU usage. If the CPU usage is influenced by those inserted logging statements, instead of the existing source code itself, our model cannot be used to identify performance-influencing source code locations to log.

Furthermore, if we identify too many locations that are statistically significantly influencing CPU usage, it is not practical for developers to log all locations nor can developers thoroughly investigate every location to ensure the need for logging. Besides, if all the identified locations are already well logged, developers may not need our approach's logging suggestion.

Approach We focus on 2.0.5 of OpenMRS, v2 of CloudStore, and one release of ES. We measure the model fit to assess the quality of the statistical models for software CPU usage. In particular, we calculate the R^2 of each model to measure model fit. If the model perfectly fits the data, the R^2 of the model is 1, while a zero R^2 value indicates that the model does not explain the variability of the dependent variable (i.e., performance metric). We also count the number of logging locations that are suggested by our approach. For every suggested logging location, we manually examine whether there already exists a logging statement.

The invocation of logging statements themselves has a performance overhead. Therefore, we measure the influence of the inserted logging statement to the fit of the model. We consider the invocation to the logging library itself as a method to monitor and create a log metric measuring the times that the logging library is called to generate logs. For every model that we built in our case study, we add the new log metric as an independent variable. By adding this independent variable into the model, we can study whether the log metric provides an increase of R^2, which represents the additional explanatory power of the execution of the inserted logging statement to the CPU usage. The increase of R^2 measures the explanatory power of the model that is provided only by the execution of the logging statements, but not the software system itself.

The *metric* that we use to assess the goal is the R^2 values of each model and the increased R^2 values from the metric of the logging statement itself.

Results Table 14 shows the model fit of every performance model built to suggest logging locations for the three subject systems. Each row of the table is a performance model. The steps in the first column of the table show that whether the model is built from which step in our approach (step 2.1 to 2.3 in Section 2). The "Web request name" and the "Method name/Block location" columns show the logging location added in each step of the performance modeling. The two columns under R^2 show the R^2 of models that are with and without considering the overhead of logging statements themselves. The difference between the values of these two columns is preferred to be small, to show that the overhead of the logging statement themselves do not impact the validity of the models.

Our statistical performance models have an R^2 of 26.9% and 90.2%. Such values of the model fit confirm that our performance models can well explain CPU usage. In addition, although we do not require a perfect model to interpret the influence of logging locations on CPU usage, a model with a very low model fit would introduce threats to the validity of our results. By looking closely at the models, we can see that the models with our automatically inserted logging statement typically has higher R^2 than the models that are only using web logs. For example, by insert logging statements into two methods in OpenMRS, the fit of the performance model almost doubles (from 26.9% to 46.3%). However, the models that are with inserted logging statements into basic code blocks have a relatively smaller increase of R^2 in comparison to the ones with method-level logging. In the same example of OpenMRS, inserting the logs into basic code blocks only provides 1.6% increase of the R^2.

Our approach does not suggest an overwhelming amount of logging locations for performance modeling. In total, our approach suggests three, two, and four locations for CloudStore, OpenMRS, and ES respectively. We consider such an amount of suggestion as an appropriate amount for practitioners. By measuring the total number of methods in the subject systems, we only suggest to log in less than 0.5% of them. By providing such suggestions to our industrial practitioners, we also received the feedback that such an amount of suggestions is not overwhelming. Hence, practitioners can allocate resource to examine each suggestion and make the final decision of whether to insert logging statements to those locations. Moreover, by manually examining each of the logging locations, we find that **none of the suggested logging locations contain logging statements.** This implies that our approach may provide additional information about the CPU usage other than what is already known by developers.

The automatically inserted logging statements' overhead does not contribute significantly to the performance models. We find that the log metric that measures the execution of the logging statements provides only little explanatory power to the models. In particular, the maximum of the increase of the R^2 is only 3.4% (see Table 14). Therefore, the inserted logging statement do not have a large impact to bias the explanatory power of our suggested logging locations.

Table 14 R^2 values of the statistical performance models built by our approach

Steps:	Web request name	Method name/Block location	R^2	
			Original	With logging statement as a metric
Cloud Store				
Step 1: Web logs only	N/A		90.20%	N/A
Step 2: With method instrumentation	”cloudstore/ ”	HomeController.getProductUrl()) (No block)	78.50%	80.50%
	”cloudstore/buy”	DaoImpl.getCurrentSession()(No block)	49.40%	49.50%
	”cloudstore/search”	ItemDaoImpl.findAllByAuthor()	78.00%	81.20%
Step 3: With block instrumentation	”cloudstore/search”	ItemDaoImpl.java, line 233 to 243	81.30%	81.60%
OpenMRS				
Step 1: Web logs only	N/A		26.90%	N/A

Step 2: With method instrumentation	concept/	ConceptServiceImpl.getAllConcepts()
		ConceptServiceImpl.getFalseConcept()
			46.30%	47.80%

Step 3: With block instrumentation	concept/	ConceptServiceImpl.java, line 300 to 302 ConceptServiceImpl.java, line 929 to 930	47.90%	48.00%
ES				
Step 1: Web logs only	N/A		43.80%	N/A

Step 2: With method instrumentation	Web request A	file1.m()
		file2.n()
			75.90%	76.40%

	Web request B	(No significance)	30.00%	N/A

	Web request C	file3.o()
		file4.p()
			42.90%	43%

TStep 3: With block instrumentation	Web request A	file1,block.r file2, block.x	70.80%	70.80%
	Web request C	file3,block.y file4, block.z	76.00%	76.30%

“No block” means that the method only has one basic block in the method body

“No significance” means that none of the methods are significant in the performance model

We only present the class names, the method names and the file names due to the limit of space, without showing the package names and the full path of the files

The logging locations suggested by our approach can help model CPU usage with a high model fit. None of those locations initially contain a logging statement.

4.3 RQ3: How Large is the Performance Influence by the Suggested Logging Locations?

Motivation In the previous research question, we find that, with our approach, we can suggest logging locations that are statistically significant for performance modeling. Even though these logging locations are statistically significant, the effect of the logging location may still be negligible. Therefore, the *goal* of this RQ is to examine the magnitude of the influence on the CPU usage by our suggested logging locations.

Approach Similar to RQ2, we focus on 2.0.5 of OpenMRS, v2 of CloudStore, and one release of ES. To understand the magnitude of the influence on the CPU usage by our suggested logging locations, we first calculate Pearson correlation between the system performance, i.e., CPU usage, and with the appearance of the suggested logging locations. Higher correlation implies that the suggested logging locations may have a higher influence on the CPU usage.

To quantify the influence, we follow a similar approach used in prior research (Shihab et al. 2011; Mockus 2010). To quantify this magnitude, we set all of the metrics in the model (each as a suggested logging location) to their median value and record the predicted CPU usage. Then, to measure the effect of every logging location, we keep all of the metrics at their median value, except for the metric whose effect we wish to measure. We increase the median value of that metric by 10%. We re-calculate the CPU usage by our prediction models after increasing the metric value by 10%. We then calculate the percentage of difference (as the *metric* to answer this RQ) caused by increasing the value of that metric. For example, if the CPU is predicted to be 60% at all metrics with median value, then after increasing the metric median value by 10%, the CPU is predicted to be 90%, the effect is 0.5, i.e., $\frac{90\% - 60\%}{60\%}$. The effect of a metric can be positive or negative. A positive effect means that a higher chance of execution the suggested logging location may increase the CPU usage. The effect implies that if we were able to control all other logging locations the same but only increase the frequency of one logging location by 10%, the performance impact will be 60% higher. This approach permits us to study metrics that are of different scales, in contrast to using odds ratios analysis, which is commonly used in prior research (Shihab et al. 2010).

Results The appearance of the suggested logging locations influences the CPU usage. Table 15 presents the influences of our suggested logging locations on the CPU usage. Each row in the table corresponds to a logging location, where the column "Web request name" and "Method name/Block location" columns indicate where the logging locations are. The rest of the columns present the influence of each logging locations. For example, the column "Pearson correlation" presents the correlation between each metric logging location and the CPU usage.

The results in Table 15 show that the appearance of the suggested logging locations typically has a strong correlation to the CPU usage. In CloudStore, all of the logging locations have a strong correlation to the CPU usage, while the correlations are moderate in OpenMRS. The relative effect shows the influence of one method while controlling all other methods. *DaoImpl.getCurrentSession()* in CloudStore has the largest effect

Table 15 The influences of our suggested logging locations on system performance (CPU usage)

Suggested logging locations		Influence	
Web request name	Method name/Block location	Pearson correlation	Relative effect (add 10%)
Cloud store			
cloudstore/	HomeController.getProductUrl()	+0.80	+1.9%
cloudstore/buy	DaoImpl.getCurrentSession()	+0.70	+12.4%
cloudstore/search	ItemDaoImpl.findAllByAuthor()	+0.87	+6.0%
cloudstore/search	ItemDaoImpl.java, line 233 to 243	+0.73	+2.5%
OpenMRS			
concept/	ConceptServiceImpl.getFalseConcept()	+0.51	+1.9%
concept/	ConceptServiceImpl.getAllConcepts()	+0.53	+2.2%
concept/	ConceptServiceImpl.java, line 300 to 302	+0.56	+2.5%
concept/	ConceptServiceImpl.java, line 929 to 930	+0.56	+2.2%
ES			
Web request A	file1.m()	−0.27	−3.4%
Web request A	file2.n()	+0.81	+11.7%
Web request C	file3.o()	−0.40	−8.0%
Web request C	file4.p()	+0.56	+4.9%
Web request A	file1, block.r	−0.26	−3.9%
Web request A	file2, block.x	+0.78	+11.1%
Web request C	file3, block.y	−0.11	−2.8%
Web request C	file4, block.z	+0.86	+8.8%

A relative positive effect means that more appearances of the logging location may result in CPU usage increase

We only present the class names, the method names and the file names due to the limit of space, without showing the package names and the full path of the files

when the appearance of the method is 10% larger than its median value: the CPU usage increases 12.4%. Table 15 shows that even method with a small effect, e.g., *ConceptServiceImpl.getFalseConcept()*, can increase the CPU usage by 1.9% if increasing its appearance by 10%.

The influence on the CPU usage may be both positive or negative. We find that some suggested logging locations in ES may have a negative influence on the CPU usage of the system, i.e., the higher the appearance of the logging location, the lower the CPU usage. By manually examining those methods, we find that these methods are related to synchronized external dependency, i.e., the invocation of these methods will cause the system to wait, leading to lower CPU usage. On the other hand, we did not find any concurrency-related anomalies, which may lead to a similar negative influence on CPU usage. Our identified cases of negative influences are often caused by executing the part of the software that is rather with light-load instead of heavy-load. By having these logs, developers can consider addressing such synchronized dependency based on how often real-life users call these methods.

Our suggested logging locations have influences on the CPU usage; while such influence can be both positive and negative.

4.4 RQ4: What are the Characteristics of the Suggested Logging Locations?

Motivation In the previous research questions, we leverage our approach to suggest logging locations to assist in performance modeling. If we can study the characteristic of these locations in the source code being performance influential, we may provide more general guidance for a developer to log similar locations in the source code.

Furthermore, prior research has proposed various techniques to provide general guidance focus logging locations on large and complex methods (Fu et al. 2014; Zhu et al. 2015), or to monitor hot methods in performance. Our approach may be of less interest if prior techniques also suggest such locations to log. The *goal* of this RQ is to examine whether our approach suggests logging locations that can also be suggested by prior research on general logging guidance.

Approach Similar to RQ 2 and 3, we focus on 2.0.5 of OpenMRS, v2 of CloudStore, and one release of ES. For each of the suggested logging locations, we manually examine the surrounding source code to understand their characteristics. In particular, the *metrics* to answer this RQ are as follows. First of all, we use the size of the source code, such as lines of code, one of the factors that prior study used to model logging decisions (Zhu et al. 2015). Moreover, uncertainty concerning control flow branches is also considered in logging decisions (Zhao et al. 2017). Therefore, we measure the source lines of code (SLOC) of the suggested methods and blocks and the cyclomatic complexity of the methods that are suggested to be logged. Furthermore, in order to identify the hot methods in the systems, we massively instrument the execution of all subject systems with JProfiler and Visual Studio Profiling tool (JProfiler 2017; Visual Studio Profiling 2017). We measure both inclusive and exclusive execution time of each method and rank all the methods by their execution time. We examine whether our suggested methods are one of the hot methods, i.e., with the highest executed time.

Results The suggested logging locations are not in complex methods. By measuring the SLOC and cyclomatic complexity, we find that the suggested logging locations are in the methods with small sizes and low complexity. The methods that are suggested to be logged have a SLOC of 4, 5 and 15 in CloudStore, and methods in OpenMRS consists of only 3 and 6 SLOC. In ES, all suggested methods have a SLOC less than 35. Similarly, the values of the cyclomatic complexity of the suggested methods in CloudStore are only 1, 2 and 2; the same values are merely 1 and 2 in OpenMRS. The small sizes and the low complexity of the methods imply that practitioner may use our approach in tandem with other approaches that are based on source code metrics.

Most of the suggested logging locations are not the performance hotspot. By examining the results of detecting hotspots using both inclusive and exclusive execution time, we find that our suggested logging locations are not typical performance hotspots. In particular, only one of the logging locations (*ItemDaoImpl.findAllByAuthor()*) is in the top 10 of hotspots in the source code (excluding methods in the library). We consider the reason is that our approach does not aim to identify the methods that are invoked often, but the ones that can explain the variance of CPU usage. Therefore, our approach may complement the detection of performance hotspots in performance assurances activities.

The suggested logging locations are typically not in complex methods nor performance hotspots. Performance engineers can use our approach to complement those traditional measurements in performance engineering activities.

4.5 RQ5: What are the Root Causes of the Suggested Logging Location Changes?

Motivation From the previous research question, we find that some of the previously suggested logging locations can no longer provide enough explanatory power to interpret performance (e.g., CPU usage) variances in our target systems. However, the reasons behind such replacements are still obscure. The *goal* of this research question is to understand the root-causes of causes of such replacement. The identified root-causes can provide more information for the practitioners to support their proactive logging decisions.

Approach In order to untangle the possible reasons behind a required logging replacement, we manually examine the code commits between two consecutive software releases if there exist a suggested replacement by our approach.

First of all, for both the existing and suggested logging locations, we review the direct source code changes within the current method or control block. However, such an intuitive approach barely provide any useful information, since our suggested logging locations usually reside in non-complex and short methods, which are rarely changed. Regardless of stability in these methods, we find that the performance influencing locations in the source code are more likely to be introduced by methods in its invocation tree. Hence, we examine the call hierarchy of method that contains the logging location and explore the derivation of CPU usage changes. This would inform us which part of the source code may potentially impact CPU usage.

Although source code changes can explain most of the variations in CPU usage, other potential factors, such as database schemas updates, can also contribute to significant deviance in CPU usage prediction. In that case, we also gather changes to all artifacts in each project made between the consecutive releases, such as configuration file changes and database updates, to synthesize a comprehensive understanding of the rationale behind those related source code changes.

Since ES is a large-scale system, there exist thousands of source code files changed between every two releases. During the evaluation process, we can hardly guarantee the overall understanding of this system due to its overwhelming size. Except for comparing the difference between two consecutive releases, especially in source code and configuration changes, we need some external assistance from developers with experience in the system. To further understand the root-causes behind the changes to logging locations, we consulted several senior developers of ES. If there still exist undetermined changes, we turn to the developers who are responsible for those specific code commits.

There is not numerical *metric* but rather the categories of root-causes that we use to answer this RQ.

Results After examining the related source code and other artifact changes, we identify the possible root-causes behind performance influencing logging locations and deprecations. The result is shown in Table 16. The columns of the table correspond to the root-causes and each row of the table corresponds to a logging location. A check mark (✓) shows that the logging location in the row is introduced due to the corresponding root-cause of the column.

Database Query Changed For database centric systems, database schema updates together with queries modifications widely exist. During the execution of a database query, the web

Table 16 Rationale behind logging statement's replacement

Log metric information			Possible root-causes of logging location suggestion or deprecation						
Project	Version	Log metric	Database query changed	Expensive data query	Conditional filter related	Repetitive invocation	Properties changes	Influence from other methods in the call graph	Utility methods
OpenMRS	2.0.5	ConceptServiceImpl.300		✓	✓		✓	✓	
		ConceptServiceImpl.929			✓		✓		
	2.0.6	PersonAttribute.111				✓	✓	✓	✓
		Format.45			✓			✓	
		Person.317				✓		✓	
		Person.601			✓	✓	✓	✓	
		Person.606			✓	✓	✓	✓	
		Patient.52		✓			✓	✓	✓
		Encounter.381						✓	✓
	2.1.2 & 2.1.3	BaseOpenmrsObject.81				✓	✓	✓	✓
	2.1.3	Concept.1027			✓		✓	✓	
		Person.186		✓			✓	✓	
ES	2018Apr.	X0						✓	✓
		X3	✓	✓					✓
		X24	✓	✓	✓				
		X43	✓		✓				
		X52	✓	✓	✓				
	2018Jun.	X14		✓		✓		✓	
		X17		✓		✓		✓	
		X28	✓	✓	✓				

server hangs and waits for responses from the database server, thus leads to a lower CPU usage on the web system. When complex query logic is changed, for example when retrieving less information after query updates, it might take database server a shorter time to respond, thus such change leads to the variance in CPU usage.

In our case studies, ES is a data-centric system with frequent updates related to its database. For example, *X3* exists inside a method that frequently retrieves and updates data in the database. However, except for the update data itself, more information that is related to this data is also eagerly fetched from the database. Such behavior is identified in prior research as one of the performance anti-patterns of database-centric systems (Chen et al. 2014).

According to the issue report that is associated with the corresponding code changes, we find these newly fetched data are used as input parameters for a function modification. With this change inside database query, the extra fetches procedure may result in the variance of CPU usage variance.

Expensive Data Query Web-based systems build an intuitive interface for users to manipulate and view their data. However, fetching a large amount of data from the database can be costly, since the resources of the web server are not fully utilized when it is waiting for the database server to respond. The importance and prevalence of I/O related performance regressions have motivated prior research to proposed techniques for automated detection of those regressions (Bezemer et al. 2014). Performance optimizing in such locations may significantly enhance system efficiency and improve the users' experience. Our approach can successfully suggest to log in methods where complex database queries with massive or expensive data manipulation are located.

Take *X24* in ES as an example of an expensive data query, this method invokes a complex SQL query. Although we admit that such a complex query is important for the system feature and sometimes cannot be avoided, such a large-scale data query may be extremely influential to the CPU usage once it is invoked. Accordingly, the method is marked as one of our suggested logging locations. Between these two releases, we find several query changes in *X24*. The code changes are with newly added access-right filtration and database structure update, which may be responsible for the update to logging decisions.

Another example of expensive data query is *ConceptServiceImpl.300*. As it is shown in the following code snippet, once this logging location is invoked, a method call will retrieve all the concepts from database and sort them according to input parameters. By default, the concepts are sorted by *conceptId*. Our logging location indicates once the method *getAllConcepts* is invoked with *sortBy* parameter value is null, it will introduce significant impact on CPU usage variance.

```
@Transactional(readOnly=true)
public List<Concept> getAllConcepts( String sortBy, boolean
    asc, boolean includeRetired) throws APIException {
   if (sortBy == null) {
      LogClass.getLog().info("ConceptServiceImpl:300");
      sortBy="conceptId";
   }
   return dao.getAllConcepts(sortBy,asc,includeRetired);
}
```

Conditional Filter Related In our approach, inserted logging statements are simply used to pinpoint source code locations, and monitor the trajectory of the system's runtime execution flow. However, if a conditional statement is introduced or modified in the source code, a new route could possibly be executed in the call graph during runtime. In that case, our prediction model together with its metrics can also be affected.

X52 and *X28* in ES are mainly responsible for fetching a certain type of special data in the database. In the new version of the system, whether to retrieve the data is determined by a newly added condition. Due to the performance impact of the data retrieval, such conditions are flagged by our approach as suggested logging locations.

In OpenMRS version 2.0.5, we notice that logging location *ConceptServiceImpl.929*, which resides in method *getFalseConcept()*, is usually invoked inside condition judgements. In other words, once this method is executed, we know the execution flow steps into a different branch that can be influential to CPU usage. We can understand the importance of such methods in predicting software CPU usage. However, its explanatory power reduces significantly after the system update. By checking the source code, we find that a newly added filtering condition may prohibit function *getFalseConcept()* and its related methods from being executed.

From the following code we can find that function *getFalseConcept()* will set a boolean concept value as a property variable. Although this value is not directly related invoked in the current method body, the condition is later accessed from other methods within the running system and causes CPU usage variances.

```
@Override
@Transactional(readOnly=true)
public Concept getFalseConcept(){
   LogClass.getLog().info("ConceptServiceImpl:929");
   if (falseConcept == null) {
      setBooleanConcepts();
   }
   return falseConcept;
}
```

Repetitive Invocation Repetitive invocation indicates methods or other code elements that locate in an iterative process. This kind of repetitive execution can significantly slow down system efficiency and prolong processing time (Sandoval Alcocer et al. 2016), depends on the complexity of logic inside the loop and number of iterations. Such a phenomenon is also studied in prior research as a *One-by-One Processing* anti-pattern (Chen et al. 2014). The following examples illustrate CPU usage variance behind logging locations with repetitive invocations.

An example for repetitive invocation would be *Person.601* and *Person.608*, both of these two locations reside in the same method *getPersonName()* in OpenMRS. In this method, all known names of a person are retrieved and iterated through two *for* loops. The first suggested logging locations reside in the first loop which gets a person's preferred name. Similarly, the second logging location resides in another loop right after the first *for* loop. The second *for* loop iterates through all names of a person. After selecting a valid preferred name from the person, the method returns a person's name only if it is not empty. In this context, we consider that this kind of iteration can explain the rationale behind these performance-influential logging locations. Similarly, *X17* from ES also follows this root-cause, where database access inside a loop is suggested as a logging location.

```
if (getNames() != null && getNames().size() > 0) {
   for (PersonName name : getNames()) {
      if (name.isPreferred() && !name.isVoided()) {
      LogPackage.LogClass.getLog().info("Person:601");
      return name;
      }
   }
   for (PersonName name : getNames()) {
      LogPackage.LogClass.getLog().info("Person:606");
      if (!name.isVoided()) {
         return name;
      }
}
```

Properties Changes Although analyzing CPU usage variances from static source code changes is feasible in some extends, the runtime information can also be important to determine source code's execution path. For example, getter and setter methods are commonly used to read and store property values, and they do not usually have a complex structure and logic inside the method body. However, the property value can possibly be used as conditions when processing some performance influential methods. In that case, properties changes are considered one of the potential performance influencing factor.

When comparing OpenMRS version 2.1.2 and 2.1.3, we find a newly suggested logging location *Concept.1027*, which locates inside a method *getRetired()*. The method decides whether a "retired" property should be added and updated. It may seem not very influential from the method itself, but we find that the value of this property is used to filter out unqualified conditions. We find that this property value is used in a performance influential logging location *ConceptServiceImpl.300*. From its call graph we notice a method *getAllConcepts()*, which fetches all sorted concepts from database. In addition, method *getAllConcepts()* takes a boolean-typed parameter "*isIncludeRetired*", which decides if retired concepts will be returned. Different SQL query command will be generated upon the "retired" property value. Therefore, in spite of being in the seemingly negligible CPU usage influence from getter and setter methods, such locations may become the root-cause of substantial CPU usage variances.

The logging locations suggested by our approach can help model CPU usage with a high model fit. None of those locations initially contain a logging statement.

Influence from other Methods in the Call Graph According to our previous finding, the suggested logging locations are located inside the non-complex and short methods. The reason these functions are selected as our target is most likely due to a complex method in its call graph. To be more specific, when our logging statement is executed, it is usually accompanied by other complicated logics or database interactions from its invocation tree.

Here we would like to take *Concept.1027* as an example again. When we trace its call graph, we find that delegated properties and requested resources are fetched and stored iteratively for all concepts. Therefore, the method that calls *Concept.1027* may be responsible for CPU usage variances. However, since this method is part of a packed RESTFul module, our logging statements can only mark its invoked method as performance influential location. The result also shows that our suggested logging locations can interpret performance influential locations that relate to external function invocations.

Another example would be *Person.186* from OpenMRS 2.1.3. The suggested logging location resides in method *getBirthDateTime*, which first fetches a person's time and date of birth, then format it according to a date-time pattern. This method is invoked when editing a person's information. All attributes of the current person's object will be copied to a newly initiated object by sequence, which makes the current method together with all related attributes resetting methods produce CPU usage variance. After comparing the source code changes between the two releases, we find that most of the changes are related to database access of the *Patient* object, which is inherited from *Person* object. Such changes may potentially influence initializing person objects and cause an impact on the CPU usage variation.

Our suggested logging locations have influences on the CPU usage; while such influence can be both positive and negative.

Utility Methods Apart from the root-causes that we list above, there is another special root-cause that associates with the utility methods. Utility methods usually indicate low-level functions that are frequently invoked and widely used across the system. If a utility method is suggested as a performance influential logging location, it would be difficult to identify the real root-causes of this kind of logging location's appearance or replacement, since the method is usually invoked by a large number of methods across the whole project. However, the extensive invocation also makes monitoring such utility methods beneficial for performance monitoring and source code optimization purposes.

For example in ES, *X0* locates at the entry of a utility method that is widely leveraged by a large number of other methods in the system. As one of the most invoked utility methods, we find hundreds of related methods in the call hierarchy graph. In addition, a large number of source code changes locates in those methods. Furthermore, we also notice that some front-end source code (like javascript files) changes may also increase uncertainty to the CPU usage.

4.5.1 Static Analysis to Suggest Logging Locations

By learning the above seven root-causes of logging location changes, one may consider leveraging static analysis to detecting such a root-cause. By examining the code and the context of each root-cause, we find that applying static analysis to accurately suggest logging location changes is challenging.

For two root-causes, i.e., *expensive data query* and *repetitive invocation*, it is not challenging to detect the root-causes in the code changes. However, not all of the cases are performance influential. For example, some database query changes may only have a negligible impact on CPU usage. In addition, prior research (Chen et al. 2014) also detects *repetitive invocation* as an anti-patterns, while the impact depends on the number of repetitions in runtime.

For one pattern, *expensive data query*, one may need to ease the detection of the root-cause with a fine-tuned heuristic. Static analysis with sub-optimal heuristic may generate a large number of false positives.

Finally, for four patterns, i.e., *conditional filter related*, *properties changes*, *influence from other methods in the call graph* and *utility methods*, it would be challenging to detect the root-cause. The dependency between the code change and the impact can be indirect or distant to each other in the call graph. Future research may investigate the possibility of detecting such root-causes.

The challenging nature of detecting these root-causes statically illustrates the need for an approach that automatically suggests logging locations for performance monitoring.

We identify seven root-causes of logging location changes, including *database query changed*, *expensive data query*, *conditional filter related*, *repetitive invocation*, *properties changes*, *influence from other methods in the call graph* and *utility methods*. We find the root-causes behind the existence and deprecation of suggested logging locations stem from a various of combined factors.

5 Discussion

In this section, we discuss the related topics based on our results.

5.1 Not all Web Requests Need Additional Logging

After applying our approach, inserting logging statements may not provide statistically significantly more explanation power to the model. For example, in the Web Request B of ES, after inserting logging statements into all associated method, none of them are statistically significant in the performance model. Such results imply that over-inserting logging statements into the source code may only provide repetitive information that is already available from other logs while leading to more noise to practitioners (Yuan et al. 2014). By looking at the web request and the methods that do not need additional logging, we find that these cases are typically simple sequential executions with low complexity. For example, *ItemDaoImpl.findAllByAuthor()* in CloudStore has a loop as an extra basic block. However, our results show that inserting logging statement into the loop would not improve the performance model. That implies that the number of iterations of the loop may not influence CPU usage significantly.

5.2 How Long do we Need to Test Performance to Suggest Logging Locations?

Performance testing is a time-consuming task (Alghmadi et al. 2016). However, our approach requires multiple iterations conducting performance tests. Even though it is straightforward to deploy the multiple performance tests in separate testing environments to reduce the time, such a solution may still be resource-costly. In order to minimize the cost of the resource, we investigate whether we may shorten the duration of the performance tests and still yield similar results.

For every performance test, we take the data from the period of the first hour, the first two hours and the first three hours. We then follow the same steps as Section 2 and examine whether we can suggest the same locations to insert logging statements. We find that we can achieve the same logging suggestions by only running one hour, two hours and three hours of the test in four, one, and six models, respectively. We need the complete four hours only in two models. This result shows that practitioners may be able to reduce the test duration in practice to receive the suggestion in a timelier manner.

5.3 Aggressiveness of Updating Logging Locations

In our case study, interestingly we find that our previously removed logging location can by suggested again. In Table 12, if we remove metric *BaseOpenmrsObject.81* due to its small effect size on CPU usage variance, the logging location will be suggested again in the new performance model. This implies that our decision on removing the logging location may be too aggressive since extra resources are needed when our approach suggests the logging location back into the source code. However, we consider this decision is a tradeoff that should be determined by the practitioners when using our approach. On one hand, not removing the logging locations that have small effect sizes may saves resources when determining the logging locations for the new version. On the other hand, the logging locations may be associated with other locations in the source code that provides more contribution to the system's performance modeling. However, since the old logging location with small effect sizes is kept in the code, it may prevent us to identify other locations that potentially be more important due to their correlations. Hence, to avoid such cases, we opt for a more aggressive decision in our case studies (see Section 4).

6 Related Work

In this section, we present the prior research related to this paper in two major topics: software performance and software logging. In particular, for the top of software performance, we discuss 1) software performance monitoring, 2) software performance modeling and 3) performance regression and benchmarks. For the topic of software logging, we discuss 1) assisting logging decisions and 2) software log evolution.

6.1 Software Performance

6.1.1 Software Performance Monitoring

There exist four typical levels of software monitoring techniques. The first, *system monitoring*, monitors the status of a running software based on the performance counters from the system. Examples of such counters include CPU usage, memory usage, and I/O traffic. Rich data from these counters are widely used to monitor system performance (Cohen et al. 2005), allocate system resources and plan capacities (Zhuang et al. 2015) or predict system crash (Cohen et al. 2004). Despite the usefulness of such data, the lack of domain knowledge of the software running on top of the system makes the data difficult to use for improving the system in a detailed level (like improving source code).

The second type of widely used techniques is based on massive *tracing*. The tracing information records every function call that is invoked during the running of the system. Prior research leverages the tracking information to system quality and efficiency (Zhang and Ernst 2014; 2015). In order to generate such tracing information, tools such as JProfiler (2017) is widely used in practice and research. The challenge of leveraging such tracing information is the extra overhead from the tracing tools. Such overhead prevents the use of tracing in a large-scale system or during the field running of the system, hence tracing is often used in the development environment by developers. Nevertheless, Maplesden et al. took advantage of patterns in tracing information. They built an automated tool to detect such patterns with the goal of improving the performance investigations and the systems' performance (Maplesden et al. 2015, 2015) .

To minimize the overhead from tracing, techniques are proposed to only trace a selected set of function calls, such that the tracing information from the field is possible to be monitored. For example, Application Performance Management tools (Ahmed et al. 2016) typically choose REST API call entry points to monitor. However, trace information is often generated automatically without the interference of developers' knowledge. The collected trace information may not all be needed for developers' particular purpose while the actually needed information may be missing.

The third type of monitoring technique is based on logging. Developers write logging statements in the source code to expose valuable information of runtime system behavior. A logging statement, e.g., *logger.info("static string"+ variable)*, typically consists of a log level (e.g., trace/debug/info/warn/error/fatal), a logged event using a static text, and variables that are related to the event context. During system runtime, the invocation of these logging statements would generate logs that are often treated as the most important, sometimes only, source of information for debugging and maintenance of large software systems. The logging information is generated based on developers' knowledge of the system, and it is flexible to monitor various information in the code. Due to the extensive value in logs, prior research has proposed to leverage logging data to improve the efficiency and quality of large software systems (Jiang et al. 2009; Chen et al. 2014; 2016; Shang et al. 2015). The advantage of using logging to monitor and analyze system performance motivates our work. In particular, with our approach, the prior research that depends on logging may benefit from the extra information that is captured from the suggested logging statements.

Finally, there exist techniques that reside in the system kernel such as JVM, to profile and monitor performance. Just-in-time (JIT) compilers are widely adopted to improve runtime performance by converting bytecode to more efficient machine code on the fly. While JIT is running, it collects profiling information for hot loops (iteration exceeds a threshold) at runtime. The profiler will trace the executions inside the loops (JIT 2018; JVM 2018). Since programs usually spend most time executing the minority amount of code, the Java Hotspot VM architecture will identify and spend most attention on performance critical parts at runtime, then avoid compiling the infrequently executed code (The Java HotSpot Performance Engine Architecture 2010). Yin et al. (2018) introduce a new approach for fine-grained methods profiling and code warm-up with low overhead. Compare to previous profilers, which usually cause significant overhead to the system or lack detailed information about the execution path, the approach uses machine code instead of byte code to expedite the profiling efficiency. The approach optimizes the code cache to perform ahead-of-time code warm-up and reduce the JIT overhead at runtime. Furthermore, the authors suggest that practitioners should optimize performance for every system according to its own application scenarios, instead of simply adopting universal solutions.

6.1.2 Performance Modeling

Performance modeling is a typical practice in system performance engineering. Due to the more complex nature of performance problems in distributed systems, simple raw metrics might not be enough. Therefore, Cohen et al. introduced the concept and use of *signatures* and *clustering* from logging data and system metrics to detect system states that are of significant impact in the system's performance (Cohen et al. 2005). With such data, Cohen et al. (2004) used TAN (Tree-Augmented Bayesian Networks) models to model the high-level system performance states based on a small subset of metrics without *a priori* knowledge of the system. Brebner et al. have application performance management (APM) data in multiple industry projects to build performance models. However, the models that depend on APM can get very complex, and customization is needed (Brebner 2016). In order to improve the quality of performance modeling and prediction. Stewart et al. (2007) consider the inconsistency of usage in enterprise and large e-commerce systems. In their work, they modeled using measurement data and *transaction mix*, and they report a better prediction quality instead of the existing scalar workload volume approach.

Since there could be too many performance metrics to be used in performance modeling, different previous researches address the issue. Xiong et al. (2013) propose an automatic creation and selection of multiple models based on different metrics. They execute tests on virtual machines using standard performance benchmarks. Shang et al. (2015) present an approach to automatically group metrics in a smaller number of clusters. They used regression models on injected and real-life scenarios, and their approach outperforms traditional approaches.

Besides the use of regression models, other statistical techniques have been used to facilitate the communication of results, such as control charts (Nguyen et al. 2012). Many different modeling approaches have been summarized by Gao et al. in three categories: rule-based models, data mining models and queueing models. In their work, they used the models to compare the effectiveness of load testing and provide insights on how to better do load testing (Gao et al. 2016). Farshchi et al. (2015) build correlation model between logs and operation activity's effect on system resources. Such correlation is later leveraged to detect system anomalies.

The rich usage of performance modeling supports our approach that leverages such model to suggest logging locations. We iteratively find the best logging locations that would provide the most significant explanatory power to the performance of the system.

6.1.3 Performance Regression and Benchmarks

In performance engineering, benchmarks can be utilized to compare the performance of various aspects of systems and their performance, such as CPUs, databases, information retrieval systems (Sim et al. 2003; Waller et al. 2015). As source code changes introduced in the software release cycle, performance regression is one of the most significant factors to assure the quality of the software system. One of the usage scenarios of these benchmarks is in the detection of performance regression.

To detect performance regression, Chen and Shang (2017) analyze over 1,000 commits from two large software systems and detect performance regression by running tests and performance micro-benchmarks repetitively, then apply a statistically rigorous approach to examine if a performance degradation occurs. Then they link the source code changes to performance regression and summarize the root-causes of performance degradation. However, their approach may not apply to software systems without mature performance testing or benchmarking infrastructure.

Van Hoorn et al. (2009) systematically introduced Kieker, a system run-time monitoring framework, to continuous monitoring and collecting system runtime behavior. This tool employs aspect-oriented programming which allows non-intrusive trace-based performance monitoring, the authors evaluate the extra performance overhead using two micro-benchmarks, the result shows implementing Kieker has low overhead to original system performance. However, application-level monitoring tools cannot pinpoint the exact source code locations which could potentially influence system performance.

Bezemer and Zaidman (2014) propose an approach based on a combination of association rules and performance counters, in order to locate bottlenecks in the system. Such locations of bottlenecks can be used as a starting point which can be later be leveraged for possible performance improvement. The evaluation of the technique on two case studies shows the high accuracy in detecting performance bottlenecks and the starting point of performance improvement is shown to be with high precision.

Waller et al. (2015) investigate regression benchmarks on performance monitoring and including performance benchmarks into continuous integration. According to their findings, the authors suggest that performance should be integrated into the software development process from the start, instead of being after releases. Furthermore, the authors also mention that the main challenge in performance regression analysis is to locate code changes that may trigger such regression. This motivates our research to investigate performance influences from the source code level.

Ahmed et al. (2016) analyze the effectiveness of Application Performance Management (APM) tools for detecting performance regression. From their study, they find commercial APM tools perform better than open-source ones in detecting performance regression.

In the first part of our approach, we leverage a modeling approach that detects the deviation of system performance from logs, which can be further leveraged as an indicator to detect performance regression.

6.2 Software Logging

6.2.1 Assist in Logging Decisions

Although logging is a significant technique for software performance monitoring, the logging practice, in general, is not as straightforward as one would expect. Logging involves a trade-off between the overhead it can generate and have the appropriate information. In previous work, Zhao et al. proposed an algorithm that touches such trade-offs. They increase the debugging assertiveness by automatically placing logs based on an overhead limit threshold (Zhao et al. 2017). Even if no overhead existed, there is still a need to balance between too much information and too little information (Fu et al. 2014).

Aiming to support the logging decisions, previous research has contributed in ways to understand, automate and suggest opportunities of where to log. Fu et al. performed an empirical study on industry systems categorizing logged snippets of code. Their work also revealed the possibility of predicting where to log according to the extracted logging features (Fu et al. 2014). Zhu et al. follow up this work and predict where to log as suggestions to developers (2015). Similarly, a tool called *Errlog* presented by Yuan et al. indicated the benefits of automatically detecting logging opportunities for failure diagnosis using exception patterns and failure reports (2012).

Previous research also presented other aspects to consider when taking logging decisions. Li et al. modeled which log level should be used when adding new logging statements (2017). In a different work, Li et al. studied log changes and modeled those log changes to provide a just-in-time suggestion to developers for changing logs (2017). Different previous research has presented what to log for a diverse set of concerns. Yuan et al. presented *LogEnhancer* that adds causally-related information to existing logging statements. Their focus was on software failures and software diagnosability (Yuan et al.

2011). Despite the above research effort, there exists no research focus on providing logging suggestions with the goal of monitoring system performance. In contrast with previous research, this paper focuses on logging suggestion for performance.

6.2.2 Software Log Evolution

Although logging statements are designed to generate important information during runtime systems, the lack of stableness of logs makes it difficult for log management and maintenance. Kabinna et al. (2016) study the logging libraries migrations based on Apache Software Foundation (ASF) projects. Their research indicates nearly 14% of those projects went through logging library changes at least once. Kabinna et al. (2016, 2018) also investigate the log stability of four open source applications and find 20 to 45 percent of the logging statements are changed at least once since created. And they also discuss the features which may impact the stability of logging statements.

Shang et al.(2011, 2014) study the evolution of logs in two open source and one industrial software. Their result shows that logging changes occur across all versions, which may potentially lead to vulnerable functionality of log processing apps. They find only 15% of the changes are unavoidable and error-prone, while the majority of logging changes are either avoidable or recoverable. They suggest system developers to avoid modifying logging statements as much as possible.

7 Threats to Validity

This section discusses the threats to the validity of our paper.

7.1 External Validity

Our evaluation is conducted on CloudStore, OpenMRS, and ES. The open source systems are adopted in prior performance engineering research studying these systems' workload (Chen et al. 2016b). The scale and the importance of ES make performance a critical matter for it and makes identifying logging locations challenging. The activeness in the development of ES makes it very costly to re-evaluate the logging locations often. Hence, part 1 of our approach is important for practitioners to adopt our approach in their actual work environment. Nevertheless, more case studies on other software in other domains are needed to evaluate our approach. All our subject systems are developed based on either Java or .Net. Our approach may not be directly applicable to other programming languages, especially dynamic languages such as Python. Further work may investigate approaches to minimize the uncertainty in performance characterization of dynamic languages.

Our approach currently only focuses on web applications. We leverage web logs in the first step in order to scope down the amount of source code to instrument. However, other researchers and practitioners may adapt our approach by applying our approach by starting on a few hot locations in the source code. Yet, without evaluation with such an approach, we cannot claim the usefulness of our approach to other types of systems.

We apply our approach to six releases of open source web-based systems. We agree that having more releases of the evaluation can further strengthen our paper. Our statistical analysis is based on a large number of data points from each version. While we agree that we can always increase the number of versions, the current setting does not impact the validity of our statistical analysis.

In this paper, we focus on pinpointing performance influential source code locations using inserted logging statements. However, external changes like configuration changes, API migrations, database structure updates, and workload variances can also potentially affect system performance. In our testing environment, we minimize the external influences by applying the same workload and identical database copy for different versions of the same branch. But these factors should still be noticed for practitioners when implementing our approach.

7.2 Internal Validity

Our approach is based on the CPU usage that is recorded by *Psutil*. The quality of recorded CPU usage can impact the internal validity of our study. Similarly, the frequency of recording the CPU usage by *Psutil* may also impact the results of our approach. Further work may further evaluate our approach by varying such frequency. Our approach depends on building statistical models. Therefore, with a smaller amount of CPU usage data, our approach may not perform well due to the quality of the statistical model. Determining the optimal amount of CPU usage data needed for our approach is in our plan. Although our approach builds statistical models using logs, we do not aim to predict nor claim the causal relationship between the dependent variable and independent variables in the models. The only purpose of building regression models is to capture the relation between logs and CPU usage.

7.3 Construct Validity

Our approach uses linear regression models to model CPU usage. Although linear regression models have been used in prior research in performance engineering (Shang et al. 2015; Xiong et al. 2013), there exist other statistical models that may model CPU usage more accurately. Our goal is not to accurately predict CPU usage but rather capture the relationship between logs and CPU usage. Further work may investigate the use of other models.

Aspect-oriented programming (AOP) is widely adopted to add behaviors without bringing extra impact on business core logic (Li et al. 2018). Using AOP frameworks to inject logs is a more convenient methodology. During our development of the toolset, we tried to use AOP instead of directly injecting logging statements. However, the open source projects (such as OpenMRS) are heavily based on their own AOP framework. Our logging AOP often have conflicts with their AOP framework and we tried to keep their existing implementation untouched as much as we can to avoid potential bias. Using the directly logging framework to insert logging statements makes the tool more applicable to be adopted. In addition, we consider that this decision would not impact our experimental results.

We chose to design our approach in an aggressive manner when deciding potential logging locations. For example, we choose a low p-value to ensure the statistical significance of the logging location. Our approach may miss potential possible logging locations. However, our goal is to prioritize on the precision of the suggestion hence making the suggestion less intrusive to practitioners. By working with our industrial collaboration, we find that a large number of logging suggestions can be overwhelming since practitioners prefer to manually verify each logging location before having actual changes to the source code.

The overhead of the logs may influence CPU usage. Although we evaluate the impact of logs on the CPU usage by examining the explanatory power of logging statements themselves, the overhead may still impact the results of our approach. Minimizing such overhead is in our further plan.

Our evaluation of our approach is based on modeling CPU usage. There exist other performance metrics, such as memory and response time, that can be modeled by logs when evaluating our approach. Also, the performance of the subject systems is recorded while running their performance tests. If a logging location is not executed by performance tests, it cannot be identified by our approach. To address this threat, we sought to use the performance test that mimics the field workload from our industrial collaborators. However, a different workload may lead to different performance influencing locations in the source code. Therefore, when applying our approach, practitioners should always be aware of the impact of the workload (the performance tests on the system). Hence, evaluation with more performance metrics and more performance tests may lead to a better understanding of the usefulness of our approach.

Although we suggest logging locations for performance assurance activities, we do not claim that they are the only relevant logging locations. Additionally, the R^2 of our models is between 26.9% and 90.2%. The R^2 shows that logs cannot explain all the variance in the CPU usage. The unexplained variance of CPU usage may due to other performance influencing source code or external influence of the system (e.g., network latency). In our future work, we plan to model other influencing factors of the CPU usage to improve our approach.

Our approach is based on automated code analysis and code manipulation when changing and rebuilding the software is needed. Such an approach may require extra resources to the system infrastructure. In our future work, we plan to alter the source code adaptively during the runtime of performance testing or in the field to improve our approach.

In our context, suggested logging locations are derived from our prediction model. Although their usefulness is validated through a statistically rigorous approach, the actual efficacy is still undetermined. Consequently, we consult several senior developers about our suggested logging locations for both versions. For all the suggested logging locations in ES, we received the confirmation on the validity of our suggestions.

8 Conclusion and Future Work

Logging information is one of the most significant sources of data in performance monitoring and modeling. Due to the extensive use of logs, all too often, the success of various performance modeling and analysis techniques often rely on the availability of logs. However, existing empirical studies and automated techniques for logging decisions do not consider the particular need for system performance monitoring. In this paper, we propose an approach to automatically suggest where to insert logging statements with the goal of support performance monitoring for web-based systems. Our approach suggests inserting logging statement into the source code locations that can complement the explanation power of statistical performance models. Moreover, our approach suggests the need for updating logging locations when system evolves. By evaluating our approach on two open source systems (CloudStore and OpenMRS) and one commercial system (ES), we find that our approach suggests logging locations that improve the statistical performance models and those suggested logging locations have a high influence on system performance (CPU usage) while not being traditional complex methods nor performance hotspots. In addition, after applying our approach on suggesting logging locations on multiple releases of our subject systems, we manually identified root-causes of logging statement suggestion and deprecation. Practitioners can integrate our approach into the release pipeline of their system to have logging suggestions periodically. In addition, the root-causes can be leveraged by researchers to prioritize their future research in suggesting logging decisions.

Future work Our future work lies in three aspects. First of all, we plan to improve our study based on more aspects of performance metrics (like memory and response time) and more types of system (like desktop systems and mobile apps). Second, we plan to apply more sophisticated modeling techniques to enhance our approach and provide more accurate logging location suggestion. Finally, we plan to investigate whether we can use data from actual users from the field to suggest logging locations.

Acknowledgments We would like to thank ERA Environmental Management Solutions for providing access to the enterprise system used in our case study. The findings and opinions expressed in this paper are those of the authors and do not necessarily represent or reflect those of ERA Environmental Management Solutions and/or its subsidiaries and affiliates. Moreover, our results do not reflect the quality of ERA Environmental Management Solutions' products.

References

The Java HotSpot Performance Engine Architecture (2010) Slowly Changing Dimensions. https://www.oracle.com/technetwork/java/whitepaper-135217.html. Oracle

.NET Compiler Platform ("Roslyn") (2017) .NET Compiler Platform ("Roslyn"). https://github.com/dotnet/roslyn

Ahmed TM, Bezemer C, Chen TH, Hassan AE, Shang W (2016) Studying the effectiveness of application performance management (apm) tools for detecting performance regressions for web applications: an experience report. In: Proceedings of the 13th International Conference on Mining Software Repositories, MSR '16. ACM, New York, pp 1–12

Alghmadi HM, Syer MD, Shang W, Hassan AE (2016) An automated approach for recommending when to stop performance tests. In: 2016 IEEE International conference on software maintenance and evolution (ICSME), pp 279–289

Apache JMeter (2015) Apache: Jmeter. http://jmeter.apache.org/. Accessed: 2015-06-01

Bezemer C, Milon E, Zaidman A, Pouwelse J (2014) Detecting and analyzing i/o performance regressions. Journal of Software: Evolution and Process 26(12):1193–1212. https://doi.org/10.1002/smr.1657. https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.1657

Bezemer CP, Zaidman A (2014) Performance optimization of deployed software-as-a-service applications. J Syst Softw 87:87–103. https://doi.org/10.1016/j.jss.2013.09.013

Bootstrap (2017) Bootstrap. https://cran.r-project.org/web/packages/bootstrap/bootstrap.pdf. Accessed: 2017-02-27

Brebner PC (2016) Automatic performance modelling from application performance management (apm) data: an experience report. In: Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering, ICPE '16. ACM, New York, pp 55–61

Chen J, Shang W (2017) An exploratory study of performance regression introducing code changes. In: 2017 IEEE international conference on software maintenance and evolution (ICSME). IEEE, pp 341–352

Chen TH, Shang W, Hassan AE, Nasser M, Flora P (2016) Cacheoptimizer: Helping developers configure caching frameworks for hibernate-based database-centric web applications. In: Proceedings of the 24th ACM SIGSOFT international symposium on the foundations of software engineering, FSE '16

Chen TH, Shang W, Hassan AE, Nasser M, Flora P (2016) Cacheoptimizer: Helping developers configure caching frameworks for hibernate-based database-centric web applications. In: Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering, FSE 2016. ACM, New York, pp 666–677

Chen TH, Shang W, Jiang ZM, Hassan AE, Nasser M, Flora P (2014) Detecting performance anti-patterns for applications developed using object-relational mapping. In: Proceedings of the 36th international conference on software engineering, ICSE 2014. ACM, New York, pp 1001–1012

Chen TH, Shang W, Jiang ZM, Hassan AE, Nasser M, Flora P (2016) Finding and evaluating the performance impact of redundant data access for applications that are developed using object-relational mapping frameworks. IEEE Trans Softw Eng PP(99):1–1

Cliff N (1993) Dominance statistics: ordinal analyses to answer ordinal questions. Psychol Bull 114(3):494

CloudScale (2017) CloudScale Project. http://www.cloudscale-project.eu/

Cohen I, Chase JS, Goldszmidt M, Kelly T, Symons J (2004) Correlating instrumentation data to system states: a building block for automated diagnosis and control. In: OSDI, vol 4, pp 16–16

Cohen I, Zhang S, Goldszmidt M, Symons J, Kelly T, Fox A (2005) Capturing, indexing, clustering, and retrieving system history. In: Proceedings of the twentieth ACM symposium on operating systems principles, SOSP '05. ACM, New York, pp 105–118

Eclipse JDT (2017) Eclipse Java development tools (JDT). http://www.eclipse.org/jdt/

Farshchi M, Schneider JG, Weber I, Grundy J (2015) Experience report: Anomaly detection of cloud application operations using log and cloud metric correlation analysis. In: 2015 IEEE 26th international symposium on software reliability engineering (ISSRE), pp 24–34

Foo KC, Jiang ZM, Adams B, Hassan AE, Zou Y, Flora P (2010) Mining performance regression testing repositories for automated performance analysis. In: 2010 10th international conference on quality software (QSIC). IEEE, pp 32–41

Freedman DA (2009) Statistical models: theory and practice. Cambridge University Press, Cambridge

Fu Q, Zhu J, Hu W, Lou JG, Ding R, Lin Q, Zhang D, Xie T (2014) Where do developers log? an empirical study on logging practices in industry. In: Companion proceedings of the 36th international conference on software engineering, ICSE companion 2014. ACM, New York, pp 24-33

Gao R, Jiang ZM, Barna C, Litoiu M (2016) A framework to evaluate the effectiveness of different load testing analysis techniques. In: 2016 IEEE International conference on software testing, verification and validation (ICST), pp 22–32

Harrell F (2001) Regression modeling strategies 2001. Springer CrossRef Google Scholar, Nashville

Van Hoorn A, Rohr M, Hasselbring W, Waller J, Ehlers J, Frey S, Kieselhorst D. (2009) Continuous monitoring of software services: Design and application of the Kieker framework

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. ACM, New York, pp 247–248

Jiang ZM, Hassan AE, Hamann G, Flora P (2009) Automated performance analysis of load tests. In: ICSM '09: 25th IEEE international conference on software maintenance

JIT (2018) Tracing just-in-time compilation. https://en.wikipedia.org/wiki/Tracing_just-in-time_compilation

JProfiler (2017) JProfiler. https://www.ej-technologies.com/products/jprofiler/overview.html

JVM (2018) Jvm jit compilation as a way of performance optimisation. https://jakubstransky.com/2018/01/15/java-jvm-jit-compilation-performance-optimisation/

Kabinna S, Bezemer C, Shang W, Hassan AE (2016) Logging library migrations: a case study for the apache software foundation projects. In: Proceedings of the 13th international conference on mining software repositories. ACM, pp 154–164

Kabinna S, Bezemer C, Shang W, Syer MD, Hassan AE (2018) Examining the stability of logging statements. Empir Softw Eng 23(1):290–333

Kabinna S, Shang W, Bezemer C, Hassan AE (2016) Examining the stability of logging statements. In: 2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER), vol 1, pp 326–337, https://doi.org/10.1109/SANER.2016.29

Kernighan BW, Pike R (1999) The practice of programming. Addison-Wesley Longman Publishing Co., Inc., Boston

Kuhn M (2008) Building predictive models in r using the caret package. Journal of Statistical Software Articles 28(5):1–26

Li H, Chen THP, Hassan AE, Nasser M, Flora P (2018) Adopting autonomic computing capabilities in existing large-scale systems: an industrial experience report. In: Proceedings of the 40th international conference on software engineering: software engineering in practice. ACM, pp 1–10

Li H, Shang W, Hassan AE (2017) Which log level should developers choose for a new logging statement? Empir Softw Eng 22(4):1684–1716

Li H, Shang W, Zou Y, Hassan AE (2017) Towards just-in-time suggestions for log changes. Empir Softw Eng 22(4):1831–1865

Log4J Async (2017) Log4J Async. https://logging.apache.org/log4j/2.x/manual/async.html

Maplesden D, von Randow K, Tempero E, Hosking J, Grundy J (2015) Performance analysis using subsuming methods: an industrial case study. In: Proceedings of the 37th international conference on software engineering - Volume 2, ICSE '15. IEEE Press, Piscataway, pp 149–158

Maplesden D, Tempero E, Hosking J, Grundy JC (2015) Subsuming methods: Finding new optimisation opportunities in object-oriented software. In: Proceedings of the 6th ACM/SPEC international conference on performance engineering, ICPE '15. ACM, New York, pp 175–186

Mockus A (2010) Organizational volatility and its effects on software defects. In: Proceedings of the Eighteenth ACM SIGSOFT international symposium on foundations of software engineering, FSE '10. ACM, New York, pp 117–126

Moore DS, Craig BA, McCabe GP (2012) Introduction to the practice of statistics. WH Freeman, New York

Nguyen TH, Adams B, Jiang ZM, Hassan AE, Nasser M, Flora P (2012) Automated detection of performance regressions using statistical process control techniques. In: Proceedings of the 3rd ACM/SPEC international conference on performance engineering, ICPE '12. ACM, New York, pp 299–310

PSUtil (2017) PSUtil. https://github.com/giampaolo/psutil

Sandoval Alcocer JP, Bergel A, Valente MT (2016) Learning from source code history to identify performance failures. In: Proceedings of the 7th ACM/SPEC on international conference on performance engineering, ICPE '16. ACM, New York, pp 37–48, https://doi.org/10.1145/2851553.2851571

Shang W, Hassan AE, Nasser M, Flora P (2015) Automated detection of performance regressions using regression models on clustered performance counters. In: Proceedings of the 6th ACM/SPEC international conference on performance engineering, ICPE '15. ACM, New York, pp 15–26

Shang W, Jiang ZM, Adams B, Hassan AE, Godfrey MW, Nasser M, Flora P (2011) An exploratory study of the evolution of communicated information about the execution of large software systems. In: 2011 18Th working conference on reverse engineering, pp 335–344. https://doi.org/10.1109/WCRE.2011.48

Shang W, Jiang ZM, Adams B, Hassan AE, Godfrey MW, Nasser M, Flora P (2014) An exploratory study of the evolution of communicated information about the execution of large software systems. Journal of Software: Evolution and Process 26(1):3–26

Shihab E, Jiang ZM, Ibrahim WM, Adams B, Hassan AE (2010) Understanding the impact of code and process metrics on post-release defects: a case study on the eclipse project. In: Proceedings of the 2010 ACM-IEEE international symposium on empirical software engineering and measurement, ESEM '10. ACM, New York, pp 4:1–4:10

Shihab E, Mockus A, Kamei Y, Adams B, Hassan AE (2011) High-impact defects: a study of breakage and surprise defects. In: Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on foundations of software engineering, ESEC/FSE '11. ACM, New York, pp 300–310

Sim SE, Easterbrook S, Holt RC (2003) Using benchmarking to advance research: a challenge to software engineering. In: proceedings of the 25th international conference on software engineering. IEEE Computer Society, pp 74–83

Stewart C, Kelly T, Zhang A (2007) Exploiting nonstationarity for performance prediction. In: ACM SIGOPS operating systems review, vol 41. ACM, pp 31–44

Syer MD, Shang W, Jiang ZM, Hassan AE (2017) Continuous validation of performance test workloads. Autom Softw Eng 24(1):189–231. https://doi.org/10.1007/s10515-016-0196-8

TPC-W (2017) TPC Benchmark W (TPC-W). http://www.tpc.org/tpcw/

Visual Studio Profiling (2017) Visual Studio Profiling. https://docs.microsoft.com/en-us/visualstudio/profiling

Waller J, Ehmke NC, Hasselbring W (2015) Including performance benchmarks into continuous integration to enable devops. ACM SIGSOFT Software Engineering Notes 40(2):1–4

Weyuker E, Vokolos F (2000) Experience with performance testing of software systems: issues, an approach, and case study. Transactions on Software Engineering 26(12):1147–1156

Xiong P, Pu C, Zhu X, Griffith R (2013) vperfguard: an automated model-driven framework for application performance diagnosis in consolidated cloud environments. In: Proceedings of the 4th ACM/SPEC international conference on performance engineering, ICPE '13. ACM, New York, pp 271–282

Yao K, B de Pádua G, Shang W, Sporea S, Toma A, Sajedi S (2018) Log4perf: Suggesting logging locations for web-based systems' performance monitoring. In: Proceedings of the 2018 ACM/SPEC international conference on performance engineering. ACM, pp 127–138

Yin F, Dong D, Li S, Guo J, Chow K (2018) Java performance troubleshooting and optimization at Alibaba. In: Proceedings of the 40th international conference on software engineering: software engineering in practice, ICSE-SEIP '18. ACM, New York, pp 11–12, https://doi.org/10.1145/3183519.3183536

Yuan D, Luo Y, Zhuang X, Rodrigues GR, Zhao X, Zhang Y, Jain PU, Stumm M (2014) Simple testing can prevent most critical failures: an analysis of production failures in distributed data-intensive systems. In: Proceedings of the 11th USENIX conference on operating systems design and implementation, OSDI'14. USENIX Association, Berkeley, pp 249–265

Yuan D, Park S, Huang P, Liu Y, Lee MM, Tang X, Zhou Y, Savage S (2012) Be conservative: Enhancing failure diagnosis with proactive logging. In: OSDI '12: Proceedings of the 10th USENIX conference on operating systems design and implementation, vol 12, pp 293–306

Yuan D, Zheng J, Park S, Zhou Y, Savage S (2011) Improving software diagnosability via log enhancement. In: ASPLOS '11: Proceedings of the 16th international conference on architectural support for programming languages and operating systems

Zhang S, Ernst MD (2014) Which configuration option should i change? In: Proceedings of the 36th international conference on software engineering, ICSE 2014. ACM, New York, pp 152–163

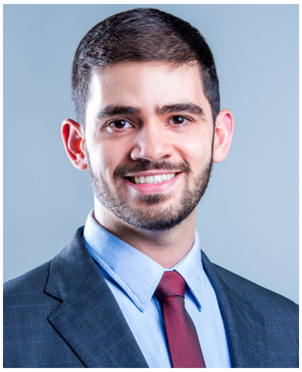
Zhang S, Ernst MD (2015) Proactive detection of inadequate diagnostic messages for software configuration errors. In: Proceedings of the 2015 international symposium on software testing and analysis, ISSTA 2015. ACM, New York, pp 12–23

Zhao X, Rodrigues K, Luo Y, Stumm M, Yuan D, Zhou Y (2017) The game of twenty questions: Do you know where to log? In: Proceedings of the 16th workshop on hot topics in operating systems, HotOS '17. ACM, New York, pp 125–131

Zhu J, He P, Fu Q, Zhang H, Lyu MR, Zhang D (2015) Learning to log: Helping developers make informed logging decisions. In: Proceedings of the 37th international conference on software engineering - volume 1, ICSE '15. IEEE Press, Piscataway, pp 415–425

Zhuang Z, Ramachandra H, Tran C, Subramaniam S, Botev C, Xiong C, Sridharan B (2015) Capacity planning and headroom analysis for taming database replication latency: Experiences with linkedin internet traffic. In: Proceedings of the 6th ACM/SPEC international conference on performance engineering, ICPE '15. ACM, New York, pp 39–50

Kundi Yao is a master student in the Department of Computer Science and Software Engineering at Concordia University, Montreal, Canada. He obtained his B.Eng. from Wuhan University of Technology, China. His research interests lie within software performance engineering, software log analytics, and mining software repositories. Contact him at ku_yao@encs.concordia.ca.



Guilherme B. de Pádua is currently a backend software developer with a constant concern for software reliability, performance and traceability through logs. He holds a Masters degree in Software Engineering at the Department of Computer Science and Software Engineering at Concordia University, Montreal, Canada. Before and during his Master's research he had already 7 years of experience in the software industry working with a variety of software systems. He obtained his B.Eng. in Computer Engineering from Universidade Federal de Itajuba, Brazil.

Weiyi Shang is an assistant professor in the Department of Computer Science and Software Engineering at Concordia University, Montreal. His research interests include big-data software engineering, software engineering for ultra-large-scale systems, software log mining, empirical software engineering, and software performance engineering. Shang received a PhD in computing from Queen's University, Canada. Contact him at shang@encs.concordia.ca.

Catalin Sporea is a senior software engineer at ERA Environmental Management Solutions. He received a Master of Science degree in Computer Engineering from the Technical University of Cluj Napoca, Romania, as well as a certificate in Information and Security Analysis from HEC Montreal. His specialization is in security of online data transactions and database structures with extensive experience in the banking industry, with a particular interest in big data applications to enterprise and industry. Contact Catalin at steve.sporea@era-ehs.com.

Andrei Toma is an analyst and project manager with ERA Environmental Solutions, in Montreal, Canada. He has worked on ERA's EH&S software development for the last 17 years and his teams successfully completed projects related to complex data structures and data processing. His major contributions are in the areas of solution concept and design, analysis and validation of proposed solutions, risk analysis, functional analysis, data analysis / data modeling and business process modeling. From time to time he enjoys taking challenges in SQL performance improvements. You can reach him at andrei.toma@era-ehs.com

Sarah Sajedi is a cofounder of ERA Environmental Management Solutions, B.Sc. in chemistry from Concordia University, Canada. She is the recipient of the Canadian Advanced Technology Alliance Sara Kirke award for innovation and corporate leadership, and was a national finalist for the RBC Women Entrepreneurs Sustainability Award for work in developing tools for manufacturers to become more sustainable as well as for implementing sustainable practices into her organization. She has been a certified ECO Environmental Professional and educator. Sajedi has been leading a team of scientists and software engineers for over twenty five years, focusing on big data, automation, and predictive analysis. Contact her at sarah.sajedi@era-ehs.com.

Affiliations

Kundi Yao[1] · Guilherme B. de Pádua[1] · Weiyi Shang[1] · Catalin Sporea[2] · Andrei Toma[2] · Sarah Sajedi[2]

Guilherme B. de Pádua
g_bicalh@encs.concordia.ca

Weiyi Shang
shang@encs.concordia.ca

Catalin Sporea
steve.sporea@era-ehs.com

Andrei Toma
andrei.toma@era-ehs.com

Sarah Sajedi
sarah.sajedi@era-ehs.com

[1] Department of Computer Science and Software Engineering, Concordia University, Montreal, QC, Canada

[2] ERA Environmental Management Solutions, Montreal, QC, Canada