Log4Perf: Suggesting and Updating Logging Locations for Web-based Systems' Performance Monitoring

Kundi Yao

A Thesis
in
The Department
of
Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of
Master of Applied Science(Software Engineering) at
Concordia University
Montréal, Québec, Canada

August 2018

Concordia University

School of Graduate Studies

This is to certify that the thesis prepared

By:	**Kundi Yao**
Entitled:	**Log4Perf: Suggesting and Updating Logging Locations for Web-based Systems' Performance Monitoring**

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science(Software Engineering)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining commitee:

Dr Tse-Hsun Chen	Chair
Dr Nikolaos Tsantalis	Examiner
Dr Jinqiu Yang	Examiner
Dr Weiyi Shang	Supervisor

Approved by

Dr Volker Haarslev, Graduate Program Director

21 August 2018

Dr Amir Asif, Dean

Faculty of Engineering and Computer Science

Abstract

Log4Perf: Suggesting and Updating Logging Locations for Web-based Systems' Performance Monitoring

Kundi Yao

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 thesis, we present Log4Perf, an automated approach that provides suggestions of where to insert logging statements with the goal of monitoring web-based systems' software performance. In particular, our approach builds and manipulates a statistical performance model to identify the locations in the source code that statistically significantly influence software performance. To evaluate Log4Perf, we conduct case studies on open source systems, i.e., CloudStore and OpenMRS, and one large-scale commercial system. Our evaluation results show that Log4Perf 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. Also, the suggested logging locations are often small and simple methods that do not have logging statements and that are not performance hotspots, making our approach an ideal complement to traditional approaches that are based on software metrics or performance hotspots. In addition, we proposed approaches that can suggest the need for updating logging locations when software evolves. After evaluating our approach, we manually examine the logging locations that are newly suggested or deprecated and identify seven root-causes. Log4Perf is integrated into the release engineering process of the commercial software to provide logging suggestions on a regular basis.

Acknowledgement

First and foremost, I am profoundly grateful to my supervisor, Dr. Weiyi Shang, for his patient guidance, encouragement, and contributive suggestions. My research would have been impossible to complete without his aid and support, and I feel extremely lucky to have an intelligent and friendly mentor who guides me in exploring innovative ideas and achieving research goals.

I would also like to show my sincere gratitude to my committee members, Dr. Nikolaos Tsantalis and Dr. Jinqiu Yang, for taking their precious time to consider my work and offer insightful comments.

Assistance provided by Guilherme B. de Pádua has been a great help in experiment setting and thesis writing. I would also like to send my gratitude to Steve Sporea, Andrei Toma, and Sarah Sajedi from ERA Environmental Management Solutions for providing access to the enterprise system used in our case study as well as valuable technical support.

I would like to send my appreciation to Dr. Weiyi Shang, Dr. Nikolaos Tsantalis, and Dr. Emad Shihab, from whom I've learned not only valuable knowledge but also the attitudes towards research, which will benefit my entire academic life. Also, I want to thank my fellow labmates from SENSE and DAS lab: Jinfu Chen, Mehran Hassani, Guilherme B. de Pádua, Maxime Lamothe, Yi Zeng, Zhenhao Li, Xiaowei Chen, Sophia Quach, Sarah Afjehei, Suihaib Mujahid, Giancarlo Sierra, Rabe Abdalkareem and Sultan Wehaibi, for the support and encouragement, also for the best moments we worked and enjoyed together.

Last but not least, I would like to send my most special thanks to my parents. I can never achieve what I have now without their love and support, being your son is the proudest thing in my life.

Related publication

Kundi Yao, Guilherme B de Pádua, Weiyi Shang, Steve Sporea, Andrei Toma, and Sarah Sajedi. Log4Perf: Suggesting Logging Locations for Web-based Systems' Performance Monitoring. In Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering, pp. 127-138. ACM, 2018.

Contents

List of Figures **viii**

List of Tables **ix**

1 Introduction **1**

2 Related work **4**

2.1 Software performance monitoring 4

2.2 Assist in logging decisions 5

2.3 Performance modeling 6

3 A motivating example **8**

4 Suggesting logging locations for a single release **10**

4.1 Approach 10

4.2 Case study setup 14

4.2.1 Subject systems and their workload 14

4.2.2 Experimental environment 16

4.3 RQ1: How well can we model system performance? 16

4.3.1 Motivation. 16

4.3.2 Approach. 17

4.3.3 Results. 17

4.4 RQ2: How large is the performance influence by the recommended logging locations? 19

4.4.1 Motivation. 19

4.4.2 Approach. 19

4.4.3 Results. 20

4.5 RQ3: What are the characteristics of the recommended logging locations? 20

4.5.1 Motivation. 20

4.5.2 Approach. 22
4.5.3 Results. 22

5 Suggesting logging locations for evolving software 23
5.1 Approach 23
5.2 Case study setup 26
5.3 RQ4: Can we suggest logging locations while software evolves? 26
5.3.1 Motivation. 26
5.3.2 Approach. 27
5.3.3 Results. 27
5.4 RQ5: What are the root causes of the suggested logging location changes? 31
5.5 Motivation 31
5.6 Approach 31
5.7 Results 32

6 Discussion 39
6.1 Performance influence from the inserted logging statement. 39
6.2 Not all web requests need additional logging. 40
6.3 How long do we need to test performance to suggest logging locations? 40
6.4 Aggressiveness of updating logging locations 41

7 Threats to validity 42
7.1 External validity 42
7.2 Internal validity 43
7.3 Construct validity 43

8 Conclusion 45

Bibliography 46

List of Figures

1 An overview of our approach to suggest logging locations for performance monitoring. 11
2 Identifying the need of updating prediction model and log metrics 24

List of Tables

1 Overview of our subject systems. 14
2 R^2 values of the statistical performance models built by our approach. 18
3 The influences of our suggested logging locations on system performance. 21
4 Details of each group of subject systems 26
5 OpenMRS Replacement decisions 28
6 ES Replacement decisions 28
7 OpenMRS log metric effectiveness evaluation 30
8 ES log metric effectiveness evaluation 30
9 Rationale behind logging statement's replacement 33

Chapter 1

Introduction

The rise of large-scale software systems, such as web-based system like Amazon, has imposed an impact on people's daily lives from mobile devices users to space station operators. 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 [WV00]. 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 [CZG$^+$05], such as collecting CPU usage, or application-level information, such as response time or throughput. In particular, Application Performance Management tools, such as Kieker [vHWH12], are widely used in practice. They collect performance data from the systems when they are running in the field environment. However, such system 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 knowledge of logs has been widely identified to improve the quality of large software systems [KP99, JHHF09, CSJ$^+$14, CSJ$^+$16, SHNF15]. Prior research proposed and used logs to monitor and improve software performance [JHHF09, CSJ$^+$14, CSJ$^+$16, SHNF15]. The success of those performance assurance techniques depends on the well-maintained logging infrastructure and the high quality of the logs. Although prior research has proposed various approaches to improve the quality of logs [ZRL$^+$17, FZH$^+$14, ZHF$^+$15, YPH$^+$12, YZP$^+$11, LSZEH17, LSH17], 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.

In this thesis, we present an approach that automatically provides logging location suggestion for web-based systems based on the particular interest in performance modeling in two parts.

Part 1: Our approach first automatically inserts logging statements into the source code. After 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.

Part 2: Software ever evolves while the locations in the source code that contributes to performance modeling also evolve. Obviously, re-applying our approach in part 1 can produce up-to-date logging locations. However, such a repetitive process not only requires extra effort but may also impact developers and operators who leverage the logs. Therefore, the second part of our approach leverage statistical analysis to evaluate whether the existing logging locations can still model system performance in a new version. In addition, our approach suggests selecting the existing logging locations that may keep in the new version.

We evaluate our approach with two open source systems, namely OpenMRS and CloudStore, and one commercial system. Our evaluation results show that we can build high-quality 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 system performance. Also, these locations cannot be identified using code complexity metrics or detected as performance hotspots. In addition, we apply our approach in suggesting logging locations on multiple releases of our subject systems. By manually examining the logging locations that are newly suggested or deprecated, we identify seven root-causes. Such root-causes can be leveraged by practitioners for proactively inserting logging statements.

This thesis 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 system performance.
- The outcome of our approach can complement the use of traditional code metrics and performance hotspots to assist performance engineers in practice.
- We identify seven root causes of suggesting or deprecating logging locations. Practitioners can leverage these root causes in their proactive logging decisions.

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 thesis is organized as follows: Chapter 2 presents the prior research that is related to this thesis. Chapter 3 presents an example to motivate our work. Chapter 4 and 5 presents our automated approach to suggest logging locations, including the results of evaluating our approach by answering three research questions. Chapter 6 discuss related topics based on the results. Chapter 7 presents the threats to the validity of our study. Finally, Chapter 8 concludes this thesis.

Chapter 2

Related work

In this chapter, we present the prior research related to this thesis in three aspects: 1) software performance monitoring, 2) assisting logging decisions and 3) software performance modeling.

2.1 Software performance monitoring

There exist three 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 is widely used to monitor system performance [CZG+05], allocate system resources, plan capacities [ZRT+15] and predict system crash [CCG+04]. 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 prove system quality and efficiency [ZE14, ZE15]. In order to generate such tracing information, tools such as *JProfiler* [EJ] are 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 [MTHG15, MvRT+15].

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 [ABC$^+$16] 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 the 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 are 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 [JHHF09, CSJ$^+$14, CSJ$^+$16, SHNF15]. The advantage of using logging to monitor and analyze system performance motivates this thesis. 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.

2.2 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 generates and having 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 [ZRL$^+$17]. Even if no overhead existed, there is still a need to balance between too much information and too little information [FZH$^+$14].

Aiming to support the logging decisions, many previous works have 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 [FZH$^+$14]. Zhu et al. follow up the work and predict where to log as suggestions to developers. Similarly, a called *Errlog* presented by Yuan et al. indicated the benefits of automatically detecting logging opportunities for failure diagnosis using exception patterns and failure reports [YPH$^+$12].

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 [LSH17]. 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 [LSZEH17]. 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 [YZP$^+$11]. 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 thesis focuses on logging suggestion for performance.

2.3 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 [CZG$^+$05]. With such data, Cohen et al. [CCG$^+$04] 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 [Bre16]. In order to improve the quality of performance modeling and prediction. Stewart et al. [SKZ07] 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 research addresses the issue. Xiong et al. [XPZG13] proposed 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. [SHNF15] presented 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 [NAJ$^+$12]. 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 [GJBL16]. Farshchi et al. [FSWG15] 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.

Chapter 3

A motivating example

Tom is a performance engineer of an e-commerce web system. He often uses the information from web logs (e.g., page requests) to build performance models to understand system performance or to detect performance issues.

Tom finds that the performance models are often unreliable in predicting the system's performance. He examines the performance of each log entry and found that some entries have a significant variance. However, there is not enough information in the web logs to accurately pinpoint the issue for further monitoring. Hence, knowing only which web requests were called is not enough to explain the performance of the system.

Let us consider an example (Algorithm 1) in which the function processes a list of products for a given signed-in customer. The products have an expiration date and, if they are expired, the program needs to consult a different supplier. In this example, the method *LoadProductStock* response time varies according to different factors, such as the number of products for that customer, and whether the products are expired or not. If the products are not expired the method might return very fast; while if the current customer has many expired products, there will be too many calls to consult suppliers, leading to the significantly long response time.

Although Tom can identify and monitor some complex requests in the web logs, he finds that some complex requests may not be so useful to monitor, since they have a steady performance behavior. For those cases, the information provided by the web logs is sufficient. Nevertheless, for the requests that their performance is not steady (e.g., Algorithm 1), there exists a high degree of uncertainty. Due to this reason, Tom needs to manually go through all the web log entries to find the scenarios (e.g., particular customer and product(s)) that required further monitoring and, therefore, require more logging statements. For a large-scale system with a non-trivial workload, this manual operation is not feasible, and, consequently, Tom needs a technique to automatically suggest

Algorithm 1 Example: Load products that has an expiration date.

```
1: function LoadProductStock($c$)
2:     $products \leftarrow$ product list of customer $c$
3:     for each $p$ in $products$ do
4:         if $IsExpired(p)$ then
5:             $p.Stock \leftarrow StockFromSupplier(p)$
6:         end if
7:     end for
8: end function
```

where the monitoring and logging are needed, without repetitive information. Such technique would significantly reduce the uncertainty of monitoring or not the right places.

In this next chapter of this thesis, we will present an approach that seeks to suggest logging locations by examining whether the location in the source code can provide significant explanatory power to the systems' performance.

Chapter 4

Suggesting logging locations for a single release

4.1 Approach

In this chapter, we present our approach that can automatically suggest 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 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 method, and we identify and suggest the code blocks where logs 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 our approach is presented in Figure 1.

Step 1: Identifying performance-influencing web requests

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

1.1 Parsing web logs

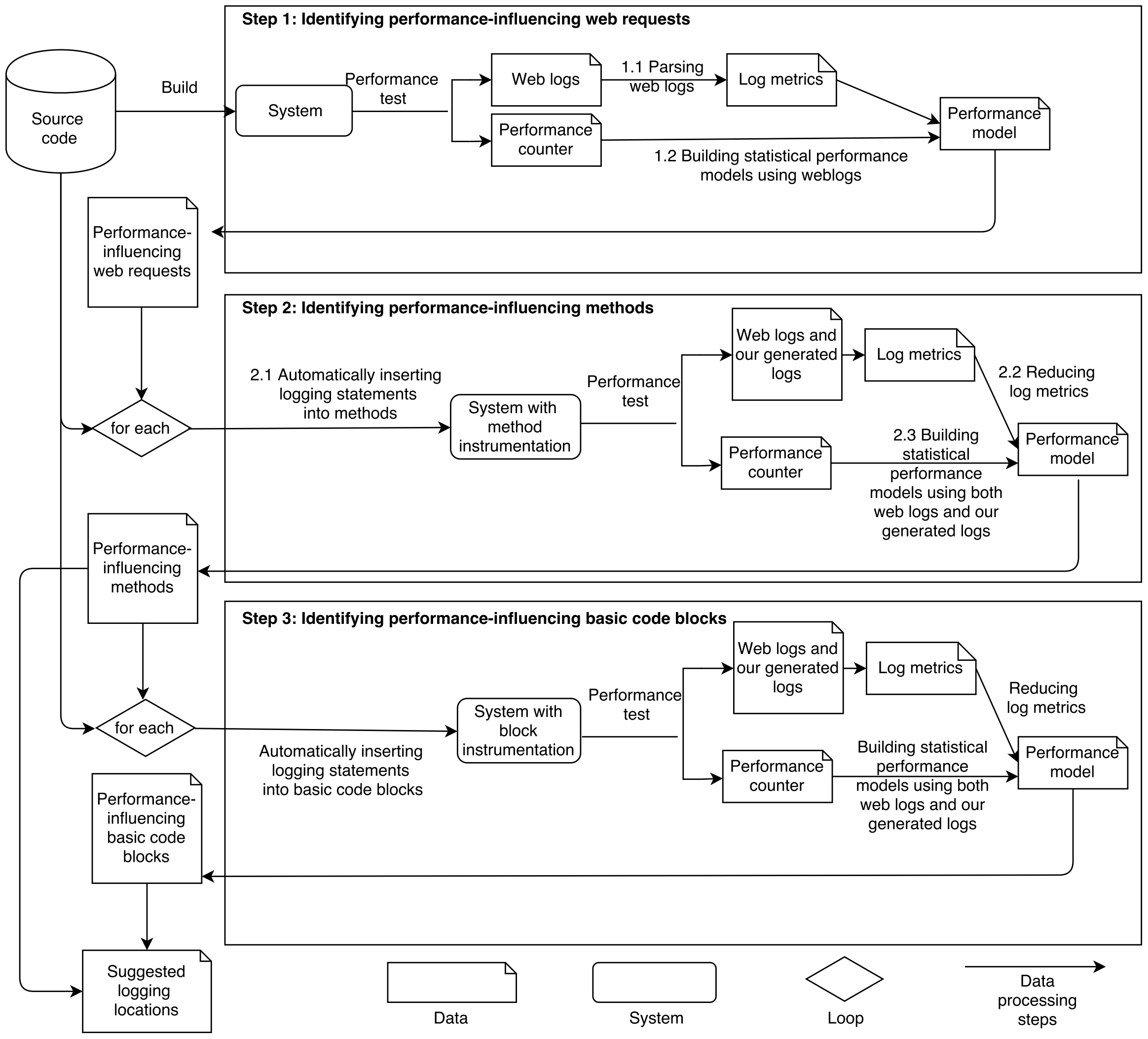

Step 1: Identifying performance-influencing web requests
Source code
Build
System
Performance test
Web logs
1.1 Parsing web logs
Log metrics
Performance counter
Performance model
1.2 Building statistical performance models using weblogs
Performance-influencing web requests
Step 2: Identifying performance-influencing methods
for each
2.1 Automatically inserting logging statements into methods
System with method instrumentation
Performance test
Web logs and our generated logs
Log metrics
2.2 Reducing log metrics
Performance counter
2.3 Building statistical performance models using both web logs and our generated logs
Performance model
Performance-influencing methods
Step 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

Figure 1: An overview of our approach to suggest logging locations for performance monitoring.

We run performance test for our subject systems and monitor their performance during the test. After the performance test, we parse the generated web logs. In particular, we keep the time stamp of the web log and the web request (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 web request being executed during the period. For example, if a web request *index.jsp* is executed 10 times during a 30-second time period, the metric *index.jsp*'s value is 10 for that period.

1.2 Building statistical performance models using web logs

We follow a model building approach that is similar to the approach from prior software performance research [SHNF15, CZG^{+}05, XPZG13]. We build a linear regression model [Fre09] to model the performance 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 [CZG+05, XPZG13, FSWG15]. We use the log metrics that are generated from web logs as independent variables. The dependent variable of the model is the performance metrics that are collected during applying the load on the software system, such as CPU usage.

After building a linear regression model for the performance 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., performance metrics. 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 log associated source code has a statistically significant influence on the software performance. Based on the list of statistically significant log metrics, we identify the performance-influencing web requests.

Step 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. 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.

2.1 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 [Ecl] and .NET Compiler Platform ("Roslyn") [Mica], to parse the source code and to identify the associated methods in the source code. We automatically insert a logging statement based on *Log4j2* and *Log4Net.Async* at the beginning of each method source code. Since the goal of our approach only suggests 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 1.1, 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.

2.2 Reducing metrics

Intuitively, methods that never execute during a workload, or the execution of the method has a constant value during the workload do not influence the performance of the system. 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 [Kuh08]. We used the Pearson correlation coefficient among all performance 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. 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 [Har01]. 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 performance 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 performance metrics with $R^2 > 0.9$, we remove the metric for method *foo*.

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 1.2. As a difference, we do not include the log metrics from web logs that are found not performance influencing from step 1.2. 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.

Step 3: Identifying performance-influencing basic code blocks

A method may be long and 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, we do not proceed with this step. For the methods with multiple basic blocks, similar to step 2.1, 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 and 2.3 to identify which code block is statistically significantly influencing performance. We then automatically suggest to developers the logging statement insertions into the basic code block, to assist in performance monitoring. If none of the log metrics that are based on basic blocks are significant, we suggest to developers the direct insertion of logging statement at the beginning of the method itself.

4.2 Case study setup

In this section, we first present the setup of our evaluation, including the subject systems, the workload, and the experimental environment. Then we evaluate our approach by answering three research questions. For each research question, we present the motivation for the question, the approach that we use to answer the question and finally the results.

4.2.1 Subject systems and their workload

We evaluate our approach with open-source software, including *OpenMRS* and *CloudStore*, and one commercial software, *ES*. The overview of the subject software is shown in Table 1.

Table 1: Overview of our subject systems.

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

OpenMRS

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 healthcare 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 [CSH$^+$16a]. *OpenMRS* provides a web-based interface and RESTFul services. We deployed the *OpenMRS* version 2.0.5 and the data used are from MySQL backup files that are provided by *OpenMRS* developers. The backup file contains data for over 5K patients and 476K observations. We use the RESTFul API test cases created by Chen et al. [CSH$^+$16a]. 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 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

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]. It was developed to validate the European Union funded project called CloudScale [Clo]. We choose *CloudStore* due to its importance in improving cloud systems performance and scalability. It has also been studied in prior research [CSH$^+$16a]. We deployed the *CloudStore* version v2 and the data used was generated using scripts provided by *CloudStore* developers. The generated data for CloudStore contains about 864K customers, 777K orders, and 300 items. We use the test cases created by Chen et al [CSH$^+$16a] to cover searching, browsing, adding items to shopping carts, and checking out. The tests include randomness to simulate real-world workloads better. For example, there is randomness to ensure that some customers may check out, and some may not. We run the performance tests with the same length as OpenMRS.

ES

ES is a commercial software that provides government-regulation related reporting services. The service is widely used as the market leader of its domain. 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.

4.2.2 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 [Apaa]. These machines have the same hardware configuration, which is 8G 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* [Gia] to monitor the performance 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 [SSJH17, ASSH16], CPU percentage of the monitored process between two timestamps are calculated as the CPU usage of the corresponding workload during the period.

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 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 [FJA+10]. We use *Log4J2*'s asynchronous logging to generate the automatically instrumented logs since it is shown to have the smallest performance overhead [Apab].

4.3 RQ1: How well can we model system performance?

4.3.1 Motivation.

The success of our approach depends on the ability to build a well fit statistical models for software performance. 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 software performance (i.e., performance metrics). Additionally, the automatically inserted logging statements have an impact on software performance. If the performance 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 performance, it is not practical for developers to log all locations nor can developers deeply 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.

4.3.2 Approach.

We measure the model fit to assess the quality of the statistical models for software performance. 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.

4.3.3 Results.

Our model can well explain system performance. Shown by Table 2, our statistical performance models have an R^2 of 26.9% and 90.2%. Such high values of the model fit confirms that our performance models can well explain system performance. 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 on whether to insert logging statements to those locations. Moreover, by manually examining each of the logging

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

Cloud Store				
Steps:	Web request name	Method name/Block location	R^2	
			Original	With logging statement as a metric
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				
Steps:	Web request name	Method name/Block location	R^2	
			Original	With logging statement as a metric
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				
Steps:	Web request name	Method name/Block location	R^2	
			Original	With logging statement as a metric
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%

Step 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.

locations, we find that **None of the suggested logging locations contain logging statements.** This implies that our approach may provide additional information about the system performance other than what is already known by developers.

The logging locations suggested by our approach significantly improve the performance models that are with a high model fit. None of those locations initially contain a logging statement.

4.4 RQ2: How large is the performance influence by the recommended logging locations?

4.4.1 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 trivial. Therefore, in this research question, we would like to examine the magnitude of the influence on system performance by our suggested logging locations.

4.4.2 Approach.

To understand the magnitude of the influence on system performance by our recommended logging locations, we first calculate Pearson correlation between the system performance, i.e., CPU, and with the appearance of the suggested logging locations. Higher correlation implies that the suggested logging locations may have a higher influence on the system performance.

To quantify the influence, we follow a similar approach used in prior research [SMK$^+$11, Moc10]. To quantify this magnitude, we set all of the metrics in the model (each as a suggested logging location) to their mean value and record the predicted system performance. 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 double the median value of that metric and re-calculate the predicted system performance. We then calculate the percentage of difference caused by doubling the value of that metric. For example, if the CPU is 60% at all metrics with median value and 90% by increasing one log metrics, 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 system performance, e.g., higher CPU usage. 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 [SJI$^+$10].

4.4.3 Results.

The appearance of the suggested logging locations influences the system performance. Table 3 shows that the appearance of the suggested logging locations typically has a strong correlation to system performance. In CloudStore, all of the logging locations have a strong correlation to 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 when the appearance of the method is double to its median value: the CPU usage increases 124%. Table 3 shows that even method with a small effect, e.g., `ConceptServiceImpl.getFalseConcept()`, can increase the CPU usage by 19% if doubling its appearance.

The influence on the system performance 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. 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 system performance; while such influence can be both positive and negative.

4.5 RQ3: What are the characteristics of the recommended logging locations?

4.5.1 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 on logging locations [FZH$^+$14, ZHF$^+$15] or to monitor hot methods in performance. Our approach may be of less interest if prior techniques also suggest such locations to log.

Table 3: The influences of our suggested logging locations on system performance.

Cloud Store			
Suggested logging locations		Influence	
Web request name	Method name/Block location	Peason correlation	Relative effect
cloudstore/	HomeController.getProductUrl()	+0.80	+0.19
cloudstore/buy	DaoImpl.getCurrentSession()	+0.70	+1.24
cloudstore/search	ItemDaoImpl.findAllByAuthor()	+0.87	+0.60
cloudstore/search	ItemDaoImpl.java, line 233 to 243	+0.73	+0.25

OpenMRS			
Suggested logging locations		Influence	
Web request name	Method name/Block location	Peason correlation	Relative effect
concept/	ConceptServiceImpl.getFalseConcept()	+0.51	+0.19
concept/	ConceptServiceImpl.getAllConcepts()	+0.53	+0.22
concept/	ConceptServiceImpl.java, line 300 to 302	+0.56	+0.25
concept/	ConceptServiceImpl.java, line 929 to 930	+0.56	+0.22

ES			
Suggested logging locations		Influence	
Web request name	Method name/Block location	Peason correlation	Relative effect
Web request A	file1.m()	-0.27	-0.34
Web request A	file2.n()	+0.81	+1.17
Web request C	file3.o()	-0.40	-0.80
Web request C	file4.p()	+0.56	+0.49
Web request A	file1, block.r	-0.26	-0.39
Web request A	file2, block.x	+0.78	+1.11
Web request C	file3, block.y	-0.11	-0.28
Web request C	file4, block.z	+0.86	+0.88

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.

4.5.2 Approach.

For each of the suggested logging locations, we manually examine the surrounding source code to understand their characteristics. In particular, the size of the source code, such as lines of code, one of the factors prior study used to model logging decisions [ZHF+15]. Moreover, uncertainty concerning control flow branches is also considered in logging decisions [ZRL+17]. 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, we massively instrument the execution of all subject systems with JProfiler and Visual Studio Profiling tool [EJ , Micb]. We measure both inclusive and exclusive execution time of each method and rank all the methods by their execution time. We would like to examine whether our suggested methods are one of the hot methods, i.e., with the highest executed time.

4.5.3 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 system performance variance. 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.

Chapter 5

Suggesting logging locations for evolving software

5.1 Approach

In this chapter, we present our approach that can automatically suggest logging locations while the software evolves. Intuitively, developers may run our approach as described in Chapter 4. However, our approach requires iterations of performance testing, which may cost extra resource for the development team, leading to the delay of having logging statements in their software.

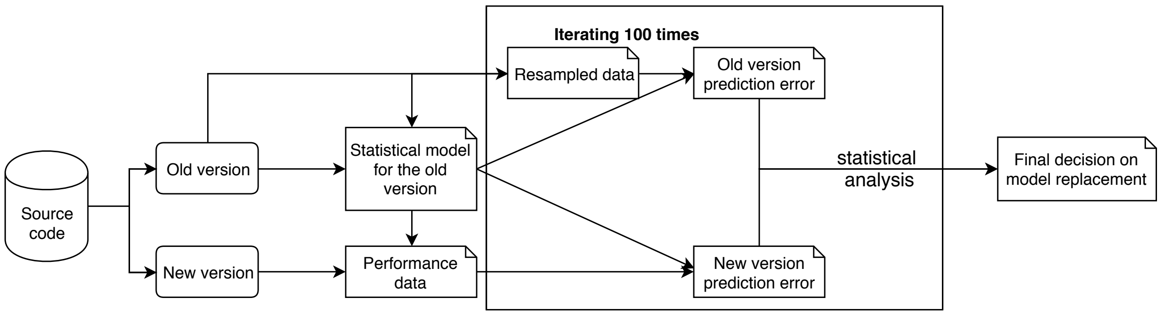
Moreover, a decision to update logging locations does not mean the every existing logging locations needs to be discarded. Simply discarding all existing logging locations is impetuous because some logging locations may still potentially influence performance, though they may not perform as significant as previous. Such logging locations should be kept as complimentary, together with new logging locations from re-applying our logging suggestion approach to the updated software system.

Therefore, we design an approach which 1) suggests whether the existing logging information needs improvement and 2) suggests which logging statements can be kept in the new version in order to minimize the iteration of performance testing.

The overview of our approach is shown in Figure 2.

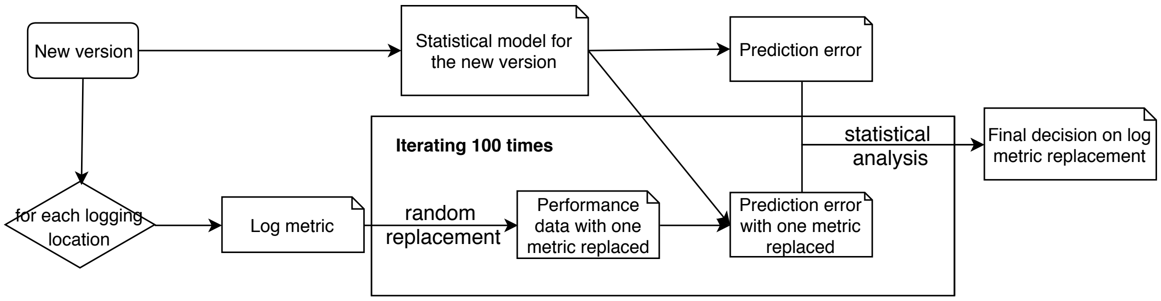
Step 1: identifying the need of updating logging locations

We aim to identify whether our existing logging locations are still effective in modeling the system's performance. If the model built from existing logging locations cannot provide an accurate performance prediction, we need to identify new logging locations for the software.



Iterating 100 times
Resampled data
Old version prediction error
Source code
Old version
Statistical model for the old version
statistical analysis
Final decision on model replacement
New version
Performance data
New version prediction error

(a) Step 1: identifying the need of updating 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 2: identifying the logging statement that can be kept in the new version

Figure 2: Identifying the need of updating prediction model and log metrics

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 performance test, we measure the performance counters, i.e., CPU usage, while collecting the logs that generated from the existing logging locations. We use the current performance model (see Chapter 4) and the logs to predict the system performance. Finally, we compare the predicted value and the actual value of the system and calculate prediction error as $|predictedCPU - actualcpu|/actualCPU$. The distribution of the prediction error of the new version represents how good is the existing logging locations and model on the new version of the system.

Intuitively, one may use a threshold (e.g., 5%) to determine whether the prediction error is too high, needing to update the logging locations. However, such thresholds may vary between systems, releases and 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 the directly using the model and data both 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 *boostrap* [Rob] to generate a new dataset from the old version to calculate the distribution of prediction error from the old version.

Finally, with the two distribution of prediction errors at hand, we compare the two distributions similar as previous studies[CSH$^+$16b], using Wilcoxon rank sum test [MCM12] and Cliff's d [Cli93]. In particular, Wilcoxon rank sum test is used to compare the mean values of prediction errors, and determine whether if 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 trivial. Therefore, we also calculate the actual effect size using Cliff's d, to further illustrate the impact size on predicted performance changes.

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

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

We repeat this process (starting from resampling the old version's data using 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 this reports to developers in order to determine the need for updating logging locations.

Step 2: identifying the logging statement that can be kept in the new version

In this step, we would like to identify the logging statements that still can help model performance in the current model and the ones that cannot. In general, we replace the values of each metric for logging statement and evaluate the impact on model prediction error. Replacing a metric that still helps model performance would significantly increase the prediction error while replacing a metric that cannot help model performance would not impact the prediction error significantly.

In particular, for each metric in the model, we replace their values by randomly selecting a value from the range between its minimum and maximum values in the original data. After replacing the 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 order to reduce the bias from our

Table 4: Details of each group of subject systems

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

metric value random replacement, we repeat this process by 100 times, similar to step 1.

5.2 Case study setup

In this section, we present the setup of our evaluation for suggesting logging locations as the software evolves. In particular, we choose the same subject systems are shown in Section 4.2. However, 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, i.e., documentation, changes between the two releases. Therefore, we do not consider 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 as two different subjects. The details of each subject system are shown in Table 4.

The experimental environment and data preprocessing for the subjects are completely the same as our previous experiments in Section 4.2. The load drivers are configured exactly the same way between releases of the same subject. In addition, in order to avoid the impact from databases, we restore the database after finishing each performance test.

5.3 RQ4: Can we suggest logging locations while software evolves?

5.3.1 Motivation.

From the previous chapter, we evaluate the impact of our suggested logging locations to performance variance. Although a source code location invocation can contribute to influencing software performance in some extends, source code changes are inevitable during the software development process. It is not realistic to expect several logging locations maintain valid indicators all the time as a panacea to performance monitoring, performance influencing locations also required to evolve as source code evolves. Thus, suggesting logging locations as software evolves is important. In this RQ, we would like to see if our proposed approach can determine the need for updating logging locations.

5.3.2 Approach.

We apply our approach to 2.0.5, 2.0.6, 2.1.0 and 2.1.3 versions of OpenMRS 2.0 and 2.1, respectively. 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 5.1). In addition, for the releases that we decide to update logging locations, we apply our approach to select the possible logging locations to keep. We examine whether the results can support us in making those decisions.

5.3.3 Results.

Our approach can clearly demonstrate the need of updating logging locations. Table 5 and 6 show our results in the need of updating logging locations. In three releases in our experiment, we make the decision of updating locations. In all of the three releases, all the predictions errors have statistically significant difference between old and new releases with all effects sizes large. On the other hand, for the releases that we decide to keep the logging locations, the majority of the prediction errors have insignificant difference, or with small or negligible effect sizes. The clear difference between the results when we decide to keep and to update logging locations demonstrates that our approach can be easily adopted by practitioners without a need for tuning thresholds. By comparing the results in Table 4, we find that between those releases where logging locations changes are recommended, 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 5: OpenMRS Replacement decisions

Group		I	II	III	IV
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

Table 6: ES Replacement decisions

Group		I	II	III
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 decided to update logging locations. We choose versions where we decide to update logging locations from Table 5 and 6. As we can see in Table 7 and 8, 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 do not produce statistically significant impact on the model. Therefore, most of the existing logging locations would not influence performance 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 error, while the effect sizes are small in most of the iterations. We first made a decision of discarding 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 fact shows us that even the small significant impact on the model is important.

Our approach can effectively suggest the need of updating logging locations. If it is decided to update logging locations, most of the existing logging locations need to be discarded.

Table 7: OpenMRS log metric effectiveness evaluation

		OpenMRS		
Group		I		IV
Version		2.0.5 –>2.0.6		2.1.2 –>2.1.3
Log Metric		ConceptServiceImpl.300	ConceptServiceImpl.929	BaseOpenmrsObject.81
# p-value >0.05		100	100	0
Effect Size & 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 8: ES log metric effectiveness evaluation

Project		ES				
Version		2018Apr –>2018Jun				
Log Metric		X52	X3	X24	X43	X0
# p-value >0.05		100	100	100	100	80
Effect Size & 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

5.4 RQ5: What are the root causes of the suggested logging location changes?

5.5 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 variances in our target systems. However, the reasons behind such replacements are still obscure. In this research question, we would like to understand the root causes of causes of such replacement. The identified root causes can provide more information for the practitioners to support their logging decisions.

5.6 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, from the perspective of already suggested logging locations, we review the direct source code changes within the current method or control block. However, the most intuitive approach barely provide contributive outcomes, 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 the performance influencing parts 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 performance changes. This would inform us which part of source code may potentially impact system performance. Similarly, we also manually check the call hierarchy starting from newly suggested logging locations, to see if an emerging log metric is introduced by source code changes.

Although source code changes can explain most of the variations in performance behavior, other potential factors, such as database schema updates, can also contribute to a significant deviance in performance 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 from 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.

5.7 Results

After examining the related source code and other artifact changes, we identify the possible root causes behind performance influencing source code locations and deprecations. The result is shown in Table 9.

Table 9: Rationale behind logging statement's replacement

Log metric information			Possible causes of logging emergence or replacement						
Project	Version	Log Metric	Database query changed	Massive data query	Conditional filter related	Repetitive invocation	Properties changes	Influence from other methods in the call graph	Utility methods
OpenMRS	2.0.5	ConceptService.300		✓	✓		✓	✓	
		ConceptService.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	✓	✓	✓				

Database query changed

For database centric systems, database schema updates together with queries modifications widely exist. When some complex query logic is changed, the influence can be negative, as we explained in RQ2. However, the CPU usage on the web server side is possibly low, due to waiting for the database server to respond, where resources are largely consumed by processing complex queries. In such a case, database queries change can potentially affect performance in the entire software system, even with a lower 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 where user-defined report template is loaded. The logging locations are executed every time when loading a specific user's preference on the components and a corresponding dynamic web form displaying user's previously saved data is generated. However, after the update, except mere preference on components, more information about the user are also eagerly fetched from the database. Such a behavior is identified in prior research as one of the performance anti-patterns of database-centric systems [CSJ$^+$14]. According to the issue report that is associated with the corresponding code changes, we find the newly fetched data is used as input parameters for a function modification. With this change inside database query, we believe the extra fetches procedure may result in the software system's performance variance.

Massive data query

Web-based systems build an intuitive interface for users to manipulate and to view their data. However, fetching a large amount of data from the database can be costly, since resources of the web server are not fully utilized when it is waiting for the database server to respond. Performance optimizing in such locations may significantly enhance system efficiency and improve the users' experience. Our approach can successfully detect methods where complex database queries with massive data manipulation are located, and these methods are proven to have an influence on system performance at runtime.

Take *X24* in ES as an example of a massive query, this method invokes a complex SQL query. The query is used to fetch previously saved reports and corresponding information from a user group. We believe such a large level data query can be extremely influential to system performance once it is invoked. Accordingly, the method is marked as one of our suggested logging locations. Between the two releases, we find several query changes in *X24*, we believe some newly added access right filtering and database structure update should be responsible for the logging deprecation.

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 to and modified in the source code, a new route could possibly be executed in the call graph during runtime. In that case, our performance prediction model together with its metrics can also be affected.

X52 in ES is mainly responsible for fetching a quick link list that contains the most commonly used links by a current user. The suggested logging locations reside in a method where user's preference on miscellaneous UI components. The widget layouts are dynamically generated after a successful login. However, from the two versions' data, we are surprised to find that the previously suggested method performs well in explaining performance variances of the system. To be more specific, this method is not even triggered in the new version. After taking a closer look at the source code and bug report, we find a recent update added restrictions to users with limited access rights. Such changes will hide visibility on documents and links from non-admin users, this will further prohibit the automated quick link list component from initializing. As a result, we consider this could be one of the major reasons for the logging replacement.

Similarly, *X28* is also influenced by newly added access right conditions. *X28* fetches the configuration information about a widget from the database, this function is used to form a dashboard for users. However, for those users who do not have a preference record in the system, there is no preset widget for users to begin within earlier versions. After the later version is released, a default widget is added to the user dashboards, accompanied with SQL queries modification and database update.

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 performance influential.We can understand the importance of such methods in predicting software performance. However, its explanatory power shrank a lot 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.

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, depending on the complexity of logic inside the loop and number of iterations. Such phenomenon is also studied in prior research as a *One-by-One Processing* anti-pattern [CSJ$^+$14].

The following examples illustrate performance variance behind logging locations with the repetitive invocation.

The most representative example of repetitive invocation would be *X17* from ES. The location resides in a method where all available items form a menu, together with their detailed information like URL fetched from the database. The method is called as part of loading users' preference. Afterward, a customized menu of the user will be generated. Inside our suggested logging location, there exists a *for* loop, where all items are reviewed and filtered. Placing logging statements in these locations can be beneficial for monitoring system's performance.

Another 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 locations reside in another loop right after the first *for* look. 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 this kind of iteration can explain the rationale behind these performance influential logging locations.

Properties changes

Although analyzing performance 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. Incidentally, we find this property value is used in another 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. In conclusion, in spite of the seemingly trivial performance influential effect from getter and setter methods, can become a substantial root cause on performance variances.

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 back in its call graph, we find delegated properties and requested resources are fetched and stored iteratively for all concepts. We believe the method that calls *Concept.1027* should be responsible for performance variances. However, since this method is part of a packed RESTful module, our logging statements can only mark its invoked method as performance influencing location. The result also shows that our suggested logging locations can interpret performance influencingl locations in related 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 performance variance.

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, *X0* locates at the entry of a method that returns a previously stored session object. As one of the most invoked utility methods, we find hundreds of related methods in the call hierarchy graph. In addition, a large amount of source code changes locate in those methods. Furthermore, we also notice that some front-end source code (like javascript files) changes may also increase uncertainty to the performance.

In OpenMRS 2.1.2, we find our suggested logging location *BaseOpenmrsObject.81* is inside a utility method names *equals()*. Method *equals()* is frequently invoked to determine whether two objects refer to the same object, or owns a same universally unique identifier(UUID) property. The method is used when interceptors are executed, including Aspect-Oriented Programming (AOP) and hibernate interceptors. Moreover, we examine the source code and find that method *equals()* are usually invoked within nest loops, which can be another factor to explain its widespread influence on performance changes.

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

Chapter 6

Discussion

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

6.1 Performance influence from the inserted logging statement.

The invocation of logging statements themselves has a performance overhead. To minimize such performance overhead, we opt to reduce the instrumentation scope at every run of the system by focusing on only one web request, web page or method at each time. Moreover, we also leveraged async-logging provided by the logging library to reduce overhead. However, introducing those logging statements still brings overhead to the system.

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 (see Table 2), 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 system performance. 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 automatically inserted logging statements do 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 2). Therefore, the inserted logging statement do not have a large impact to bias the explanatory power of our suggested logging locations.

6.2 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, whiling leading to more noise to practitioners [YLZ+14]. 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 performance significantly.

6.3 How long do we need to test performance to suggest logging locations?

Performance testing is a time-consuming task [ASSH16]. However, our approach requires multiple iterations of 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 Chapter 4 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 more timely manner.

6.4 Aggressiveness of updating logging locations

In our case study, interestingly we find that our previously removed logging location can by suggested again. In Table 7, if we remove metric *BaseOpenmrsObject.81* due to its small effect size on performance 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 5.3).

Chapter 7

Threats to validity

This chapter discusses the threats to the validity of our study.

7.1 External validity

Our evaluation is conducted on CloudStore, OpenMRS and ES. All subject systems have years of history and there is prior performance engineering research studying these systems' workload [CSH+16a]. 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 for 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 application. 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 on other types of systems.

In this thesis, we focus on pinpointing performance influencing 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 try to minimize the influences by applying same testing load and identical database for different versions. But these factors should still be noticed for practitioners when implementing our approach.

7.2 Internal validity

Our approach is based on the system performance that is recorded by *Psutil*. The quality of recorded performance can impact the internal validity of our study. Similarly, the frequency of recording system performance 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 performance data, our approach may not perform well due to the quality of the statistical model. Determining the optimal amount of performance 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 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 system performance.

7.3 Construct validity

Our approach uses linear regression models to model system performance. Although linear regression models have been used in prior research in performance engineering [SHNF15, XPZG13], there exist other statistical models that may model system performance more accurately. Our goal is not to accurately predict system performance but rather capture the relationship between logs and the system performance. Further work may investigate the use of other models.

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 system performance. Although we evaluate the impact of logs on system performance 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 system 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 from 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 system performance. The unexplained variance of performance 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 system performance 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 performance 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, since they process abundant experience understanding and contributing to the system, and we received confirmation on the validity of our suggested locations.

Chapter 8

Conclusion

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 thesis, 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. 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 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 learned by practitioners to assist in proactively putting logging statement in their source code for performance monitoring purposes.

Bibliography

[ABC$^+$16] Tarek M. Ahmed, Cor-Paul Bezemer, Tse-Hsun Chen, Ahmed E. Hassan, and Weiyi Shang. 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, pages 1–12, New York, NY, USA, 2016. ACM.

[Apaa] Apache. Jmeter. `http://jmeter.apache.org/`. Accessed: 2015-06-01.

[Apab] Apache. Log4J2 Async. `https://logging.apache.org/log4j/2.x/manual/async.html/`. Accessed: 2017-10-09.

[ASSH16] H. M. Alghmadi, M. D. Syer, W. Shang, and A. E. Hassan. An automated approach for recommending when to stop performance tests. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 279–289, Oct 2016.

[Bre16] Paul Charles Brebner. 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, pages 55–61, New York, NY, USA, 2016. ACM.

[CCG$^+$04] Ira Cohen, Jeffrey S Chase, Moises Goldszmidt, Terence Kelly, and Julie Symons. Correlating instrumentation data to system states: A building block for automated diagnosis and control. In *OSDI*, volume 4, pages 16–16, 2004.

[Cli93] Norman Cliff. Dominance statistics: Ordinal analyses to answer ordinal questions. *Psychological bulletin*, 114(3):494, 1993.

[Clo] CloudScale. CloudStore. `https://github.com/CloudScale-Project/CloudStore/`. Accessed: 2017-10-09.

[CSH+16a] Tse-Hsun Chen, Weiyi Shang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. 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, pages 666–677, New York, NY, USA, 2016. ACM.

[CSH+16b] Tse-Hsun Chen, Weiyi Shang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. 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, 2016.

[CSJ+14] Tse-Hsun Chen, Weiyi Shang, Zhen Ming Jiang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. Detecting performance anti-patterns for applications developed using object-relational mapping. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 1001–1012, New York, NY, USA, 2014. ACM.

[CSJ+16] T. H. Chen, W. Shang, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora. Finding and evaluating the performance impact of redundant data access for applications that are developed using object-relational mapping frameworks. *IEEE Transactions on Software Engineering*, PP(99):1–1, 2016.

[CZG+05] Ira Cohen, Steve Zhang, Moises Goldszmidt, Julie Symons, Terence Kelly, and Armando Fox. Capturing, indexing, clustering, and retrieving system history. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, pages 105–118, New York, NY, USA, 2005. ACM.

[Ecl] Eclipse. Eclipse Java development tools (JDT). `http://www.eclipse.org/jdt/`. Accessed: 2017-10-09.

[EJ] EJ Technologies. JProfiler. `https://www.ej-technologies.com/products/jprofiler/overview.html/`. Accessed: 2017-10-09.

[FJA+10] King Chun Foo, Zhen Ming Jiang, Bram Adams, Ahmed E Hassan, Ying Zou, and Parminder Flora. Mining performance regression testing repositories for automated performance analysis. In *Quality Software (QSIC), 2010 10th International Conference on*, pages 32–41. IEEE, 2010.

[Fre09] David A Freedman. *Statistical models: theory and practice*. cambridge university press, 2009.

[FSWG15] M. Farshchi, J. G. Schneider, I. Weber, and J. Grundy. 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)*, pages 24–34, Nov 2015.

[FZH+14] Qiang Fu, Jieming Zhu, Wenlu Hu, Jian-Guang Lou, Rui Ding, Qingwei Lin, Dongmei Zhang, and Tao Xie. 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, pages 24–33, New York, NY, USA, 2014. ACM.

[Gia] Giampaolo Rodola. psutil. `https://github.com/giampaolo/psutil/`. Accessed: 2017-02-02.

[GJBL16] R. Gao, Z. M. Jiang, C. Barna, and M. Litoiu. A framework to evaluate the effectiveness of different load testing analysis techniques. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 22–32, April 2016.

[Har01] FE Harrell. Regression modeling strategies. 2001. *Nashville: Springer CrossRef Google Scholar*, 2001.

[JHHF09] Zhen Ming Jiang, Ahmed E. Hassan, Gilbert Hamann, and Parminder Flora. Automated performance analysis of load tests. In *ICSM '09: 25th IEEE International Conference on Software Maintenance*, 2009.

[KP99] Brian W. Kernighan and Rob Pike. *The Practice of Programming.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[Kuh08] Max Kuhn. Building predictive models in r using the caret package. *Journal of Statistical Software, Articles*, 28(5):1–26, 2008.

[LSH17] Heng Li, Weiyi Shang, and Ahmed E. Hassan. Which log level should developers choose for a new logging statement? *Empirical Softw. Engg.*, 22(4):1684–1716, August 2017.

[LSZEH17] Heng Li, Weiyi Shang, Ying Zou, and Ahmed E. Hassan. Towards just-in-time suggestions for log changes. *Empirical Softw. Engg.*, 22(4):1831–1865, August 2017.

[MCM12] David S Moore, Bruce A Craig, and George P McCabe. *Introduction to the Practice of Statistics.* WH Freeman, 2012.

[Mica] Microsoft. .NET Compiler Platform ("Roslyn"). `https://github.com/dotnet/roslyn/`. Accessed: 2017-10-09.

[Micb] Microsoft. Visual Studio Profiling Tools. `https://docs.microsoft.com/en-us/visualstudio/profiling/`. Accessed: 2017-10-09.

[Moc10] Audris Mockus. Organizational volatility and its effects on software defects. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 117–126, New York, NY, USA, 2010. ACM.

[MTHG15] David Maplesden, Ewan Tempero, John Hosking, and John C. Grundy. Subsuming methods: Finding new optimisation opportunities in object-oriented software. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, ICPE '15, pages 175–186, New York, NY, USA, 2015. ACM.

[MvRT$^+$15] David Maplesden, Karl von Randow, Ewan Tempero, John Hosking, and John Grundy. Performance analysis using subsuming methods: An industrial case study. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ICSE '15, pages 149–158, Piscataway, NJ, USA, 2015. IEEE Press.

[NAJ$^+$12] Thanh H.D. Nguyen, Bram Adams, Zhen Ming Jiang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. Automated detection of performance regressions using statistical process control techniques. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, ICPE '12, pages 299–310, New York, NY, USA, 2012. ACM.

[Rob] Rob Tibshirani. Bootstrap. `https://cran.r-project.org/web/packages/bootstrap/bootstrap.pdf/`. Accessed: 2017-02-27.

[SHNF15] Weiyi Shang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. 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, pages 15–26, New York, NY, USA, 2015. ACM.

[SJI$^+$10] Emad Shihab, Zhen Ming Jiang, Walid M. Ibrahim, Bram Adams, and Ahmed E. Hassan. 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, pages 4:1–4:10, New York, NY, USA, 2010. ACM.

[SKZ07] Christopher Stewart, Terence Kelly, and Alex Zhang. Exploiting nonstationarity for performance prediction. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 31–44. ACM, 2007.

[SMK$^+$11] Emad Shihab, Audris Mockus, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. 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, pages 300–310, New York, NY, USA, 2011. ACM.

[SSJH17] Mark D. Syer, Weiyi Shang, Zhen Ming Jiang, and Ahmed E. Hassan. Continuous validation of performance test workloads. *Automated Software Engineering*, 24(1):189–231, 2017.

[TPC] TPC Benchmark W (TPC-W). `http://www.tpc.org/tpcw/`. Accessed: 2015-06-01.

[vHWH12] André van Hoorn, Jan Waller, and Wilhelm Hasselbring. 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, pages 247–248, New York, NY, USA, 2012. ACM.

[WV00] E.J. Weyuker and F.I. Vokolos. Experience with performance testing of software systems: issues, an approach, and case study. *Transactions on Software Engineering*, 26(12):1147–1156, Dec 2000.

[XPZG13] Pengcheng Xiong, Calton Pu, Xiaoyun Zhu, and Rean Griffith. 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, pages 271–282, New York, NY, USA, 2013. ACM.

[YLZ$^+$14] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm. 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, pages 249–265, Berkeley, CA, USA, 2014. USENIX Association.

[YPH$^+$12] Ding Yuan, Soyeon Park, Peng Huang, Yang Liu, Michael M Lee, Xiaoming Tang, Yuanyuan Zhou, and Stefan Savage. Be conservative: Enhancing failure diagnosis with proactive logging. In *OSDI '12: Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, volume 12, pages 293–306, 2012.

[YZP$^+$11] Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. Improving software diagnosability via log enhancement. In *ASPLOS '11: Proc. of the 16th international conference on Architectural support for programming languages and operating systems*, 2011.

[ZE14] Sai Zhang and Michael D. Ernst. Which configuration option should i change? In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 152–163, New York, NY, USA, 2014. ACM.

[ZE15] Sai Zhang and Michael D. Ernst. Proactive detection of inadequate diagnostic messages for software configuration errors. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 12–23, New York, NY, USA, 2015. ACM.

[ZHF$^+$15] Jieming Zhu, Pinjia He, Qiang Fu, Hongyu Zhang, Michael R. Lyu, and Dongmei Zhang. Learning to log: Helping developers make informed logging decisions. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 415–425, Piscataway, NJ, USA, 2015. IEEE Press.

[ZRL$^+$17] Xu Zhao, Kirk Rodrigues, Yu Luo, Michael Stumm, Ding Yuan, and Yuanyuan Zhou. 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, pages 125–131, New York, NY, USA, 2017. ACM.

[ZRT$^+$15] Zhenyun Zhuang, Haricharan Ramachandra, Cuong Tran, Subbu Subramaniam, Chavdar Botev, Chaoyue Xiong, and Badri Sridharan. 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, pages 39–50, New York, NY, USA, 2015. ACM.