Modeling Management Metrics for Monitoring Software Systems

by

Miao Jiang

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2011

© Miao Jiang 2011

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Software systems are growing rapidly in size and complexity, and becoming more and more difficult and expensive to maintain exclusively by human operators. These systems are expected to be highly available, and failure in these systems is expensive. To meet availability and performance requirements within budget, automated and efficient approaches for systems monitoring are highly desirable. Autonomic computing is an effort in this direction, which promises systems that self-monitor, thus alleviating the burden of detailed operation oversight from human administrators. In particular, a solution is to develop automated monitoring systems that continuously collect monitoring data from target systems, analyze the data, detect errors and diagnose faults automatically. In this dissertation, we survey work based on management metrics and describe the common features of these current solutions. Based on observations of the advantages and drawbacks of these solutions, we present a general solution framework in four separate steps: metric modeling, system-health signature generation, system-state checking, and fault localization. Within our framework, we present two specific solutions for error detection and fault diagnosis in the system, one based on improved linear-regression modeling and the second based on summarizing the system state by an information-theoretic measurement. We evaluate our monitoring solutions with fault-injection experiments in a J2EE benchmark and show the effectiveness and efficiency of our solutions.

Acknowledgments

I would like to express my sincere gratitude to my academic advisor, Professor Paul A.S. Ward, for allowing me to pursue my research interests and for providing continuous guidance and financial support throughout my doctoral studies. I am grateful to my PhD committee members, Dr. Lin Tan, Dr. Johnny Wong, Dr. Andrew Heunis, and Dr. Miroslaw Malek for their effort in evaluating this work and for their recommendations for improving it.

I am thankful to Mohammad Ahmad Munawar for his help in improving various aspects of this work. I am grateful to colleagues and faculty members of the Network and Distributed System Laboratory, in particular the Shoshin Laboratory, for their assistance and enriching discussions.

There are many other people who have helped me directly or indirectly for studies or otherwise during my time at the University of Waterloo – Thank you all!

Contents

Abstract iii

List of Tables ix

List of Figures x

1 Introduction 1

1.1 System Monitoring by Modeling Metrics . . . 2

1.2 Thesis Contributions . . . 3

1.3 Thesis Organization . . . 4

2 Background 6

2.1 Terminology . . . 6

2.2 Management Metrics . . . 8

2.3 Component-Based Distributed Software Systems . . . 9

2.3.1 The Java Platform, Enterprise Edition . . . 10

2.3.2 Monitoring Infrastructure . . . 12

2.4 Modeling Techniques . . . 15

2.4.1 Linear Regression . . . 15

2.4.2 Information Entropy . . . 17

3 Related Work 20

3.1 System Monitoring Based on Linear Regression Models . . . 21

3.2 System Monitoring Based on Other Linear Models . . . 22

3.3 System Monitoring Based on Non-linear Models 23
3.4 Fault Diagnosis 25
3.4.1 Fault Diagnosis by Supervised Learning 27

4 Problem Definition **29**
4.1 Error Detection 30
4.1.1 Measurement of Detection Quality 31
4.2 Fault Diagnosis 33
4.2.1 Measurement of Diagnosis Quality 35

5 Solution Framework **38**
5.1 A General Solution Framework for System Monitoring 38
5.2 Metric Modeling 39
5.2.1 Specific-Form Modeling 40
5.2.2 General-Form Modeling 42
5.3 System-Health Signature Generation 43
5.4 System-State Checking 43
5.5 Fault Localization 45

6 Solution One: Linear Models **47**
6.1 Problems of Simple Linear Regression 47
6.1.1 Heteroscedasticity 47
6.1.2 Varying Coefficients 52
6.1.3 Multi-variable Correlations 54
6.2 Improving Simple Linear Regression 55
6.2.1 Detecting Non-constant Error Variance 55
6.2.2 Generalized Least Squares 56
6.2.3 Fitness Score for Confidence Intervals 58
6.3 System-Monitoring Solution 59
6.3.1 Metric Modeling 60

6.3.2 System-Health Signature Generation 61
6.3.3 System-State Checking 61
6.3.4 Fault Localization 64

7 Solution Two: Information-Theoretic Models 65

7.1 Approach Overview 65
7.2 Computing Similarities between Metrics 66
7.3 Metric Modeling by Clustering Correlated Metrics 67
7.3.1 Identifying Correlated Metrics 68
7.4 Tracking Groups of Related Metrics 69
7.4.1 Observations on Cluster Entropy 71
7.4.2 Error Detection by Wilcoxon Rank-Sum Test 72

8 Evaluation 73

8.1 Evaluation Approach 73
8.1.1 Methodology 79
8.1.2 Fault Injection 79
8.1.3 Fault-Injection Experiments 82
8.2 Evaluation of Linear Modeling 83
8.2.1 The Performance of Individual Models 83
8.2.2 Error-Detection Examples 84
8.2.3 Experimental Results 88
8.2.4 Comparison with Prior Work 89
8.3 Evaluation of Information-Theoretic Solution 91
8.3.1 Identifying Non-linear Correlations 91
8.3.2 Clustering of Metrics 94
8.3.3 Error-Detection Examples 96
8.3.4 Experimental Results 96
8.4 Computational Cost 101

9 Conclusion and Future Work **104**

9.1 Future Research Work 105

Appendix **106**

A Addressing Specific Problems in Linear Modeling **107**

A.1 Modeling Varying Coefficients 107

A.2 Modeling Three-Variable Correlation 108

A.3 System Monitoring with New Models 108

A.3.1 Metric Modeling 110

A.4 Evaluation 111

A.4.1 Error-Detection Results 113

A.4.2 Understanding RLS Performance 114

B Limitations of Fault Localization with Metric Correlations **118**

B.1 Simplified View of a System of Correlations 120

B.1.1 Causation and Correlation 120

B.1.2 Cluster of Correlations 121

B.1.3 Effects of Invalid Causality 123

B.2 Realistic Model of a System of Correlations 125

B.2.1 Cluster of Correlations 126

B.2.2 Effects of Invalid Causality 127

References **130**

List of Tables

6.1 Regression coefficients varying with a third variable 54

7.1 Correlations captured by r^2 and NMI 68

8.1 Examples of metrics collected 78

8.2 Summary of the faults injected 80

8.3 Fault parameters 82

8.4 Model performance definition 84

8.5 Model performance 84

8.6 Error detection and fault localization with linear models 90

8.7 Error-detection comparison 92

8.8 Error-detection summary 92

8.9 Error detection with information-theoretic models 99

8.10 Error detection with different NMI thresholds 100

8.11 Computational cost 102

A.1 Error-detection summary 114

List of Figures

2.1 Overview of a Java EE-based architecture 11
2.2 Monitoring infrastructure of a Java EE-based system 12

4.1 The relationship between models, metrics, and subsystems 34

6.1 Heteroscedasticity example 48
6.2 Confidence intervals using OLS regression 49
6.3 Heteroscedasticity example: varying coefficients 50
6.4 Heteroscedasticity example: inaccurate linear model 52
6.5 Varying coefficients of a metric-pair model 53
6.6 Confidence intervals using GLS regression 58
6.7 Fitness score calculation 60
6.8 Model learning and system monitoring 61
6.9 Learning metric-correlation models 62

7.1 Approach overview 66
7.2 Metric similarity measures: r^2 and NMI 69

8.1 Experimental setup 75
8.2 Overall structure of the Trade application 76
8.3 Sample fault detection - Mis-ds-authentication 86
8.4 Sample fault detection - Mis-ds-connection-pool 86
8.5 Sample fault detection - Del-AccountJSP 87
8.6 Sample fault detection - Del-DisplayQuoteJSP 87

8.7 Sample fault detection - DB-QuoteEJB 88

8.8 Metric-similarity measures 93

8.9 Metric relationship - a power function 94

8.10 Metric relationship - a piecewise function 95

8.11 Sample in-cluster entropy 96

8.12 Sample in-cluster entropy 97

8.13 Sample in-cluster entropy 97

8.14 Sample in-cluster entropy 98

8.15 Sample in-cluster entropy 98

A.1 Model learning and system monitoring 110

A.2 Learning metric correlation models 111

A.3 Sample fault detection - a simple case 112

A.4 Sample fault detection - tolerating invalid models 113

A.5 Sample error-detection results 116

A.6 OLS models *vs.* RLS models 117

B.1 Fault localization example 119

B.2 Metrics' clusters 122

B.3 The effects of invalid causality 123

B.4 Realistic view of correlations by m_0 126

B.5 Realistic view of the effects of a fault 127

Chapter 1

Introduction

Enterprise software systems are at the core of business activity, requiring high availability and good performance. They have become critical to the success of many businesses. These systems have grown in size and complexity, while they are still expected to be highly available, offer good performance, and operate within constrained budgets. Failure in these systems is expensive, as it may cause downtime, loss of sales, customer dissatisfaction, *etc.* However, as software is not perfect and fault-protection mechanisms are not always present, system failures do occur. The major types of failures include unavailable systems, exceptions and access violations, incorrect answers, data loss and corruption, and poor performance [67].

To meet availability and performance requirements, businesses rely on human operators to actively monitor this critical infrastructure, identifying system errors and failures, diagnosing faulty components, and restoring the system to a correctly functioning state. However, as the size and complexity of software systems rapidly grows, this solution becomes more and more expensive. For example, it is expected that network, computer systems, and database administrators, as well as computer-system analysts, will be some of the fastest growing occupations from 2004 to 2014 [25]. In particular, the number of human administrators/operators in America will grow by 35% in ten years, exceeding 1.1 million by 2014 [25]. This tendency is confirmed with a new projection from 2008 to 2018 [46]: database administrators are expected to grow by 20.3%, network and computer systems administrators are expected to grow by 23.2%. In addition, this solution is not necessarily effective. System complexity can easily overwhelm administrators, affecting their ability to identify and resolve problems promptly [44]. In fact, many system failures are caused by operator errors. In some studies it is reported that 40% of failures are due to operator error [67, 83].

In addition, many approaches for monitoring software system require knowledge of system structure and details. For example, queuing models require knowledge of system structure and dynamics (*e.g.*, [85]). Likewise, some fault models require knowing all possible faults and component dependencies (*e.g.*, [87]). The required information is not always available, and modeling is a difficult manual task. In addition, as systems evolve, keeping the models up-to-date requires considerable manual effort.

In sum, as the size and complexity of software systems increase, manual intervention to address problems in these systems is becoming difficult and error prone [44]. In addition to the larger number of sophisticated components, the interactions and inter-dependencies among the many components are more dynamic and harder to comprehend. Therefore, it is essential to find automated and efficient approaches to systems monitoring. Autonomic computing [44], also known as self-managing systems, is an effort in this direction, which promises systems that self-monitor, therefore alleviating the burden of detailed operation oversight from human administrators. The idea of self-managing systems has received much attention both from the research community and the industry (*e.g.*, see [6, 18, 26, 51]), and this dissertation is intended to augment that literature.

1.1 System Monitoring by Modeling Metrics

Software systems increasingly offer a large amount and variety of monitoring data to use in system monitoring, including log records, performance metrics, traces, *etc.* While in principle the availability monitoring data can help in system monitoring, error detection, and problem determination, in practice useful information is often hard to find quickly in a sea of data. A solution to this problem is to develop automated monitoring systems that continuously collect monitoring data from target systems, analyze the data, and report when the behaviour of some of the data deviates from what is expected.

In our work we focus on metric data because of its richness and the ease of its collection. Software systems typically expose management metrics that reflect their behaviour, performance, and state. Examples of these metrics include operation invocation counts, response time, resource utilization, *etc.* Often these metrics can be collected on demand. An important advantage of metrics is that their collection costs less than alternatives such as traces [56]. Metrics are aggregate measures computed in place, whereas traces contain timestamps and fine-grained data on

system operations, and thus are expensive to collect.

In previous studies [22, 34, 62, 63, 64], people have found that a correctly functioning enterprise-software system exhibits long-term, stable correlations between many of its monitoring metrics. These correlations are expected to hold during normal operating conditions. When errors occur in the system, some of the correlations may no longer hold, potentially enabling error detection and fault localization. With this approach models can be built in an un-supervised way and without requiring knowledge of system structure. Once models are built and learned from a correctly functioning system, they can be used to classify metric samples collected during monitoring to detect errors. Further, such classifications may be extended to localize the faulty component, thus helping diagnosis.

The benefits of system monitoring by metrics analysis include:

- Metrics are easily collected on demand.
- No knowledge of the system structure or its dynamics is required beforehand, therefore it is easy to apply the technique to a wide variety of systems, which potentially makes the technique scalable.
- Error detection is automated, and human work for diagnosis may be reduced given information provided by monitoring.
- Computational overhead can be kept low with efficient modeling techniques.

1.2 Thesis Contributions

This work presents a solution to the problem of monitoring the health of complex software systems and localizing the faults that manifest in these systems. Specifically, this dissertation makes the following novel and significant contributions:

- We devise a solution framework for error detection and fault localization in complex software systems. The solution is based on the idea of relationship modeling of management metrics and is generally applicable for most complex software systems with management-metrics collection mechanisms.
- We create one solution for the problem based on the modeling of linear relationships between management metrics, which is the most frequently observed relationship in management metrics.

- We create a second solution based on an information-theoretic modeling of general relationships between management metrics. This solution can potentially capture all categories of relationships, while being much more computationally efficient compared with previous solutions.

- We experimentally validate our solutions using a realistic test-bed and a wide range of faults. We show that our approach is very effective in detecting many different errors, and significantly better than prior approaches.

- Our linear solution is about five-times faster than the prior approaches, though it is still $O(n^2)$, where nn is the number of metrics being monitored. Our information-theoretic solution is about two orders-of-magnitude faster compared with the linear solutions and, more importantly, $O(n)$.

1.3 Thesis Organization

This dissertation is organized as follows:

- **Chapter 2** introduces the basic terminology we use in the dissertation, as well as basic information needed to understand this dissertation. In particular, it covers the basics of management metrics and distributed software systems, followed by introductions to basic modeling techniques involved in this dissertation.

- **Chapter 3** is a brief survey of the prior research in systems monitoring, error detection, and fault diagnosis.

- **Chapter 4** defines the problem of interest that we solve in this dissertation.

- **Chapter 5** presents our solution framework for the system monitoring. It is also contains examples of related prior work to illustrate how the prior work fits our solution framework.

- **Chapter 6** presents our linear-modeling solution for system monitoring. It starts with observations of the shortcomings of some previous work and refines the modeling to better reflect the behaviors of management metrics in software systems.

- **Chapter 7** presents our information-theoretic solution for system monitoring. This solution is both more general and more efficient compared with the linear-modeling solution.
- **Chapter 8** experimentally evaluates solutions we created. It also contains descriptions of our experimental setup and our evaluation methodology.
- **Chapter 9** concludes our work and discusses some other related work we have done and outlines our future work in system monitoring.

Chapter 2

Background

Before we propose our formal definition of error detection and fault diagnosis based on management metrics, and our general four-step solution framework for the problem, we provide some background information in this chapter and present a brief survey on current work in this area in Chapter 3.

We first introduce the terminology we use in this thesis, and then provide information about metrics and how they are collected. An introduction to the J2EE platform follows, since we use such a platform to evaluate our work. The thesis of Munawar [65] has a very good introduction to these topics; therefore we reproduce the relevant material in Section 2.1, 2.2 and 2.3 to make this thesis self-contained. Then we provide a review of modeling techniques in Section 2.4.

2.1 Terminology

The terminology used throughout this thesis follows that of Avizienis *et al.* [2]. For completeness, we reproduce the relevant definitions.

- A *system* is an entity that interacts with other entities (*i.e.*, other systems such as software, humans, the physical environment, *etc.*). These other entities define the *environment* of the given system. A system is composed of a set of *components* put together in order to interact, where each component is another system. This recursive definition stops when further decomposition is either not possible or not of interest.

- The total state of a system is the set of the states of its components. The *behavior* of a system is a sequence of states through which the system implements its function.

- The *structure* of a system is what enables it to generate its behavior.

- The *service* delivered by a system is its behavior as it is perceived by its user(s). The part of the system boundary where service delivery takes place is the service *interface*. The part of the system's total state that is perceivable at the service interface is its *external state*; the remaining part is its *internal state*.

- The *function* of a system is what it is intended to do and is described by the *functional* specification in terms of functionality and performance.

- A *service failure* is an event that occurs when the delivered service either does not comply with the functional specification, or when the specification did not adequately describe the system function.

- An *error* is the part of the total state of the system that may lead to its subsequent service failure.

- A *fault* is the cause of an error.

- A *partial failure* occurs when a subset of several functions implemented by the system fails; the system still offers services that have not failed to the user(s). A component failure represents a fault for its parent system and from the perspective of interacting components [48].

In addition to the standard definitions above, we use the following terminology throughout this thesis.

- A *model* is a description of some characteristics of a system that can be used to study or predict those characteristics.

- An *anomaly* is a departure or deviation from the normal or the expected characteristics as determined by a model. It is important to note that anomalies do not always reflect errors or failures in a system, they may also happen because of normal, albeit uncommon, events (*e.g.*, a sudden change in user behavior).

- The *health* of a system is the degree to which its observed behavior and performance conform with the expected behavior and performance.
- *Monitoring* is the act of observing a system for the purpose of ensuring that certain properties are maintained. In our case the purpose is to make sure that the system is free of errors and failures. We use the terms monitoring and error detection interchangeably.
- *Diagnosis* is the process of identifying causal factors underlying some observed anomaly. We use the terms diagnosis, problem determination, fault localization, and root-cause analysis interchangeably.
- The *target system* is the system to be monitored.
- A *monitoring system* is the entity that monitors the target system. A monitoring system is often part of a larger *managing system*, whose role extends to other system-management functions.

2.2 Management Metrics

A *management metric* is a variable measuring an attribute or a parameter of a managed entity. An attribute either represents an instantaneous property of the monitored entity (*e.g.*, free-memory size) or an aggregation of the underlying measure over a specified time interval (*e.g.*, CPU utilization).

Metrics differ according to the scale in which they are measured. A variable with *nominal* or *categorical* scale takes values from a set of exclusive, unordered values (*e.g.*, male/female). A variable with *ordinal* scale takes a value from a set of exclusive, ordered values (*e.g.*, low/medium/high). We can determine the relative order of the values, but the difference between any two values is undefined. A variable with *interval* scale takes values for which differences can be computed. However, the values start from an arbitrary point (*i.e.*, there is no notion of a zero value). Temperature measured in Fahrenheit is an example for an interval-scale variable. A *ratio* variable is similar to an interval variable with the added property that zero means that the underlying attribute or parameter is nil (*e.g.*, travel speed). Our work focuses on metrics which have an interval or a ratio scale; these metrics represent the majority of metrics exposed by software systems.

Management frameworks such as the Simple Network Management Protocol (SNMP) [8] refine the classification of metrics. In SNMP, for example, a *counter* is a

non-negative integer that increments to a maximum and rolls over to zero. A *gauge*, on the other hand, is a variable that can increase or decrease subject to a minimum and a maximum. In addition, it is not necessary for the measurement of a metric to only be described by a single numeric value. The measurement may be represented as an object with several attributes. The Java Enterprise Edition Management Specification [74] defines various types of objects to represent performance data. A `TimeStatistic` object, for example, reports the number of times an operation occurs, the total time taken for the occurrences, and the minimum and maximum times observed.

Metric measurements are recorded in variables which may be read and updated either by the managed or the managing entity. The monitoring logic or instrumentation that updates these variables is often part of the system structure. In cases where such instrumentation does not exist, it is possible to statically or dynamically instrument components of a software system.

Management frameworks such as SNMP [8] and JMX [80] specify encoding, transport protocols, and mechanisms to collect metric measurements. In general, two mechanisms exist to collect the metrics. A managing entity can use polling (pull mechanism) to read the variables when needed. Alternatively, the managed entity can send notifications (push mechanism) containing the measurements to the managing entity.

2.3 Component-Based Distributed Software Systems

To facilitate development and enable scalability, software systems for network-based services are typically built using component-based frameworks. Many standards for implementing component-based distributed systems exist, including Common Object Request Broker Architecture (CORBA) [66], Java Platform Enterprise Edition (Java EE) [79], Distributed Component Object Model (DCOM) [52], and .Net [50]. These frameworks allow components of the same system to be distributed across different machines. These frameworks entail the use of middleware that takes care of issues such as remote communication, data exchange, object naming, registration, discovery, object life-cycle management, security, *etc.*

These component-based software systems are typically organized in tiers, each addressing specific needs. For example, a basic system to support an online store

includes a data tier comprising a database management system for persisting data, a business logic tier comprising an end-user application and an application server providing the execution environment for the application, and a presentation tier comprising an HTTP server and other software to render results of service invocations. In addition, each tier may be hosted on separate machines, each running its own operating system.

2.3.1 The Java Platform, Enterprise Edition

One of the most popular frameworks to implement distributed, component-based software systems is Java EE. The experimental aspect of this work only involves Java EE; nevertheless, we believe that the insights that our work provides extend to the other component-based frameworks.

Java EE specifies application program interfaces (APIs) and interactions for basic services needed for distributed and enterprise computing. It also defines interfaces, roles, and deployment details of components in the framework. A simple Java EE-based system is illustrated in Figure 2.1. A Java EE server is a runtime environment for executing Java EE applications. It consists of component containers, which take care of the components' lifecycle, thread management, concurrency control, resource pooling, replication, access control, *etc.* It also implements various common services and libraries. A Java EE server allows the execution of multiple applications or many instances of the same application concurrently. Many such servers exist on the market, *e.g.*, IBM WebSphere, BEA WebLogic, Oracle Application Server, JBoss, and Jonas.

A Java EE application is a combination of many specialized components. A typical Java EE application can be accessed via its web interface by making HTTP requests, by using native Java calls, or by employing other means such as web-service calls. On the server side, HTTP requests for dynamic content are handled by web components such as Java Servlets or Java Server Pages (JSP), which are managed by a *web container*. The application logic concerned with the processing of business data is implemented in Enterprise Java Beans (EJBs). These EJBs can be accessed using a remote method invocation (RMI) protocol. The Java EE specification classifies EJBs into three different types. A *session bean* is a component that acts temporarily on behalf of a client. This component can be stateful (*e.g.*, keeping track of a customer's shopping cart) or it can be stateless (*e.g.*, only computing a formula given some input). An *entity bean* is an EJB that provides a mapping to persistent data, typically a row in a database table. A *message-driven*

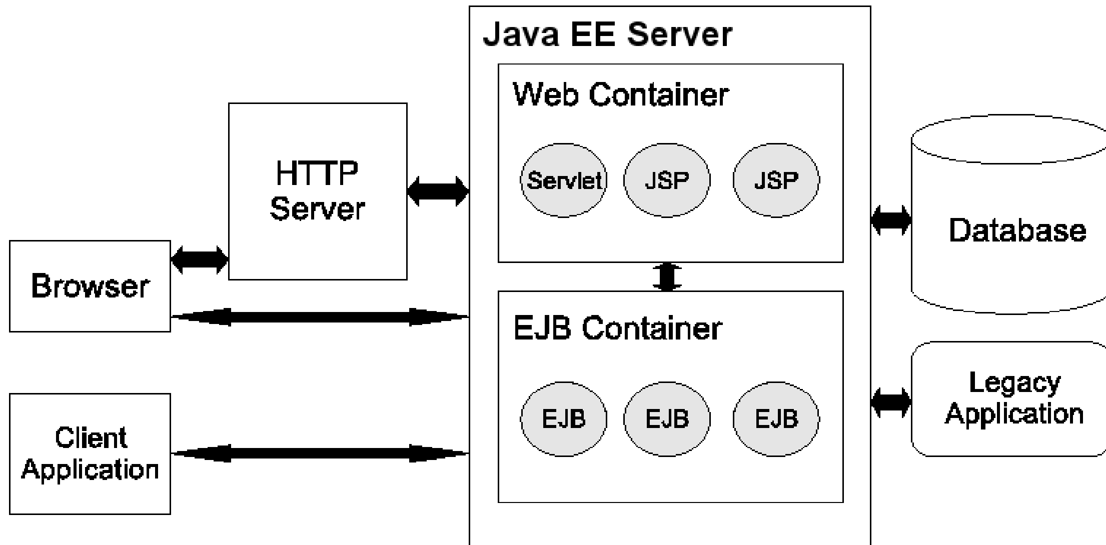


Figure 2.1: Overview of a Java EE-based architecture

bean allows an application to provide asynchronous functionality. For example, such a component can accept a customer order, adding it to a queue of pending orders; when resources become available, the orders are removed from the queue for processing. Web components and enterprise beans execute in containers, which provide the linkage between components and services and functionality implemented by the underlying runtime. Java EE applications typically require connection to back-end data sources, which may include database servers or legacy systems.

Servicing user requests in a typical Java EE-based system entails processing by many components of different types. A typical flow of execution may include the following: a client requests a service through a web page; the request is assigned to a thread at the server, which executes a Servlet. The Servlet code retrieves a reference to a Session EJB component and executes one of its methods; the Session EJB causes one or more Entity EJBs to either be instantiated or fetched; the data mapped to the Entity EJBs is retrieved by using a connection to the back-end database; once the data is fetched at the session EJB, it is processed, and then returned to a JSP component; in the JSP, the results are put in HTML format and sent to the client. While servicing the request, the components involved may utilize common services such as transactions or logging.

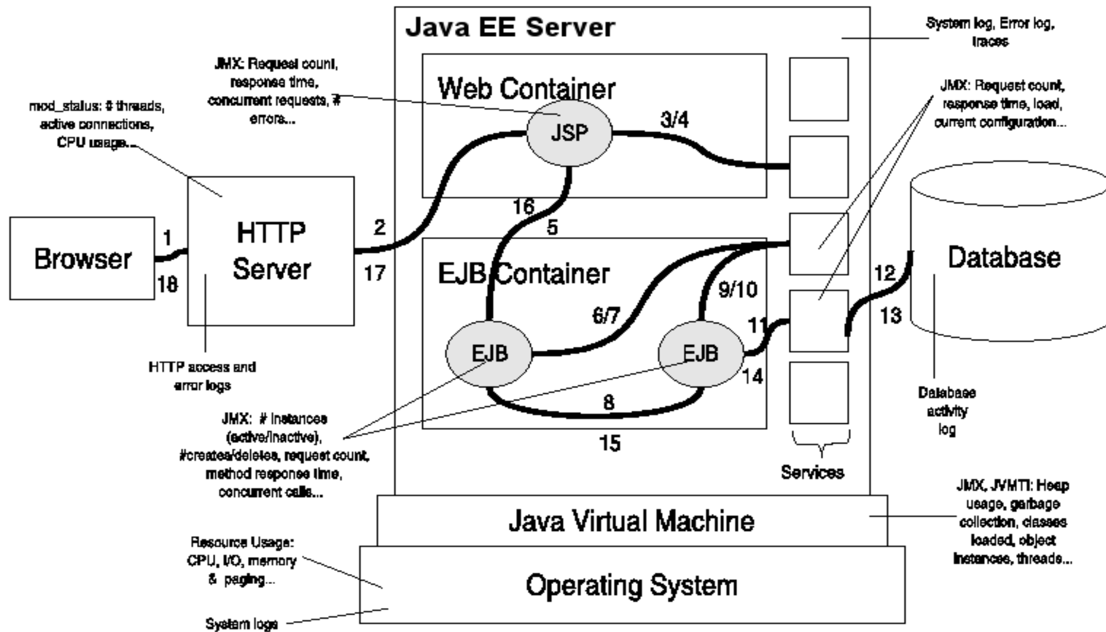


Figure 2.2: Monitoring infrastructure of a Java EE-based system

2.3.2 Monitoring Infrastructure

Software systems expose much data to enable their monitoring and management. Each subsystem can be monitored via a multitude of metrics and events, each detailing some aspect of its state, behavior, or performance. Much of the available data can be accessed through predefined mechanisms such as logging, tracing, or polling of management interfaces. Additional data can be collected on-demand at runtime by instrumenting parts of the system. Monitoring a software system, therefore, entails dealing with potentially large volumes of data. A glimpse of the amount of the data available can be illustrated by considering the monitoring infrastructure of a basic Java EE-based system. Figure 2.2 presents an overview of some important sources of information available from various parts of such a system. Below, we describe the main subsystems, the type of data they provide, and how such data can be collected.

A software system requires an operating system to function. When distributed, multiple operating systems support the software system. Most commodity operating systems provide mechanisms and tools to monitor resource usage, user activity, process behavior, *etc.* In Unix, for example, metrics are exposed through a virtual file system mounted at `/proc`. Utilities such `ps`, `vmstat`, `iostat`, and `netstat` make access to the data even more convenient. Similarly, the Windows Manage-

ment Instrumentation (WMI) [53] allows for the monitoring of many aspects of a system when using Windows. Besides these conventional monitoring facilities, much more data can be collected via dynamic instrumentation [7, 55, 81] and dynamic insertion of interceptors between components via hot-swapping [72].

Software systems commonly rely on runtime environments executing above the operating system layer. These runtimes not only make it possible to develop portable software but also implement features to improve robustness and performance. Examples of these features include sandboxing, automatic memory management and exception handling, runtime code optimization and replacement, *etc.* Such runtimes include the Java Virtual Machine (JVM) [75] and Microsoft's Common Language Runtime (CLR) [54]. A Java EE-based system requires a JVM to execute. The JVM provides different interfaces for monitoring. The JVM Tool Interface (JVMTI) [76] enables debugging as well as profiling of Java applications. A JVM can also be monitored via a standardized management interface, namely the Java Management Extensions (JMX) [77] interface. JMX allows data related to various aspects of the JVM, including the number and state of threads, memory usage, classes instantiated, and garbage collection to be accessed easily. The JMX technology is much more generic, as it provides a common management interface for Java applications to make monitoring data available and expose configuration interfaces. It also defines a scalable notification-based architecture for monitoring. In addition, it is possible to instrument Java bytecode dynamically at runtime (see, *e.g.*, [78]). Monitoring probes that were not considered at design and implementation time can now be retrofitted when the need arises. The availability of runtime bytecode instrumentation in the JVM allows Java applications to take advantage of approaches like dynamic aspect-oriented programming (see, *e.g.*, [32]), whereby monitoring aspects can be added dynamically. This represents another potential source of monitoring data.

Most Java EE-based systems require a database management system (DMBS) to manage persistent data. These DBMS expose a rich set of monitoring data to facilitate their tuning and maintenance (see, *e.g.*, [27]). Examples of the available data include details on query execution, table activity, application connections, I/O, threads, memory, storage, and locking.

Java EE applications are typically accessed via their web front-end. As such, HTTP servers are the first subsystems to handle user-requests. They usually serve static content (*e.g.*, images) directly, but redirect requests for dynamic content to an application server. They may also provide authentication and encryption services. HTTP servers also make state, performance, and error-related data available

through log files or monitoring interfaces. An HTTP server usually logs requests received, return codes, execution time, *etc.* It is also possible to query the server's state (*e.g.*, to find the number of active worker threads, number of connections alive, CPU usage per worker thread, *etc.*). For example, the `mod_status` module [1] of the Apache HTTP server provides a mechanism for collecting such data.

The application server lies at the center of a Java EE-based system, as it provides the middleware and the runtime environment to execute the application logic. Significant events (*e.g.*, exceptions) which occur during a server's execution are typically logged or sent in the form of notifications to registered listeners. There is a wide range of state, performance, and error-related data that can be collected by querying provided interfaces (*e.g.*, see [31]). Most Java EE servers are JMX-enabled [80], which allows a management entity to monitor and manage them. Many subsystems of a Java EE-based system may be shipped with embedded instrumentation that makes more detailed information available on a per-request basis (*e.g.*, using the ARM API [43]).

A Java EE server is itself organized into multiple subsystems, which include component containers (*e.g.*, web and EJB) and modules for transactions management, database connection management, thread pool and object pool management, *etc.* Each such subsystem exposes data related to the state, behavior, and performance of the subsystem. A Java EE application and its components can also make fine-grained monitoring data available. Because of standardization, much monitoring data related to applications is generic (*i.e.*, applies to all applications that conform to the Java EE specification). Still, application-specific monitoring can be made available by instrumenting the application. Data on web components, such as Servlets, may comprise the number of requests being served over time or at any time instant, number of errors encountered, response time, *etc.* As with EJBs, depending on the type of bean, different aspects can be observed. For example, one could monitor how many instances of each bean type have been created, the number of active beans, the number of free beans available in various pools, average response time per bean, the number of times the various methods of a bean are called, *etc.* For entity beans, which are usually mapped to table rows, one could check the number of times bean data is stored to or loaded from the database and the time taken for storing or loading the bean. Similarly, for message beans, one could keep track of the number of messages handled by the bean. Data as detailed as the time taken by a particular remote method of an EJB can be collected.

As illustrated above, even a basic Java EE-based system can produce a large amount of monitoring data. A few hundred metrics may be available from the appli-

cation server and the DBMS for an application such as an online store. Production-level Java EE-based systems are generally larger and more complex, comprising clustered web and application servers, replicated databases, load balancers, *etc.* Effectively monitoring such systems is very challenging. The difficulty lies in using the data generated by these systems to good effect; that is, for quickly detecting errors and failures and for localizing their causes. Furthermore, collecting all this data would not only adversely affect performance, but would create significant overhead for handling the collected data. An important aspect of the challenge is to contain this overhead, while not sacrificing effectiveness of problem determination.

2.4 Modeling Techniques

In this section we introduce the basics of the modeling techniques we used in system monitoring with management metrics.

2.4.1 Linear Regression

Linear regression find the best estimation of a target variable given other explanatory variables by assuming there is linear relationship between these variables. Given a set of pair of values $\{x_i, y_i : i = 0..n\}$, the linear regression model for the two variable is thus:

$$y_i = \hat{y}_i + \epsilon_i \tag{2.1}$$

$$\hat{y}_i = \beta_0 + \beta_1 x_i \tag{2.2}$$

where x and y are the two variables assumed correlated, and the $\beta = (\beta_0, \beta_1)$ are the model parameters. y_i is considered as the sum of its theoretic value $\hat{y}_i$ and an error ϵ_i. The errors, ϵ_i, is assumed to be independent and identically normally distributed.

Finding the parameter of best fit can be done by ordinary least squares regression, which find β's estimation $\hat{\beta}$ such that the sum of squared residuals $\sum (y_i - \hat{y}_i)^2$ is minimized.

The solution to this optimization is easily computed with analytical formulas:

$$\hat{\beta}_1 = \frac{\sum(x_i - \overline{x})(y_i - \overline{y})}{\sum(x_i - \overline{x})^2} \tag{2.3}$$

$$\hat{\beta}_0 = \frac{1}{n}(\sum y_i - \beta_1 \sum x_i) \tag{2.4}$$

where

$$\overline{x} = \frac{\sum_{i=0}^{n} x_i}{n} \tag{2.5}$$

and

$$\overline{y} = \frac{\sum_{i=0}^{n} y_i}{n} \tag{2.6}$$

The goodness of fit can be measured using the *coefficient of determination* R^2, which represents the proportion of variance in the dependent variable that is captured by the model. This measure is computed by:

$$R^2 = 1 - \frac{\sum(y_i - \hat{y}_i)^2}{\sum(y_i - \bar{y})^2}$$

R^2 can be shown to be the same as the square of a linear similarity measure Pearson product-moment correlation coefficient, $r(X, Y)$, which measures the strength of the linear relationship between X and Y:

$$r(X, Y) = \frac{\mathrm{cov}(X, Y)}{\sigma_x \sigma_y} \tag{2.7}$$

$$cov(X, Y) = E((X - E(X))(Y - E(Y))) \tag{2.8}$$

$$\sigma_x = \sqrt{cov(X, X)} \tag{2.9}$$

$$\sigma_y = \sqrt{cov(Y, Y)} \tag{2.10}$$

where $E(\dot{})$ is the expectation, $\mathrm{cov}(X, Y)$ is the covariance of the random variables X and Y. σ_x and σ_y are the variances respectively. Because $r(X, Y)$ ranges from -1 to 1, its square (*i.e.*, R^2) is usually adopted as a similarity measure.

2.4.2 Information Entropy

In this section we provide a brief overview of the concepts of information entropy involving our monitoring.

The information entropy introduced by Shannon [71] measures the uncertainty or unpredictability of a random variable. For a discrete random variable X, the entropy is given by:

$$\begin{aligned} H(X) &= E_p \ln \frac{1}{p(X)} \\ &= -\sum_{i=1}^{n} p(x_i) \ln p(x_i) \end{aligned} \tag{2.11}$$

where X takes values from the set $\{x_1, x_2, ..., x_n\}$, E_p refers to the expectation with respect to the probability distribution of X characterized by the density function p. If $p(X = x_i) = 1$ and $p(X = x_j) = 0$ for any $i \neq j$, *i.e.*, there is no uncertainty about X, then $H(X)$ is zero. Otherwise, $H(X)$ takes a positive value. $H(X)$ is at its maximum when all the outcomes x_i are equally likely.

Conditional entropy measures the uncertainty of a random variable Y given another random variable X. It represents the remaining uncertainty of Y knowing values taken by X. It is defined by:

$$\begin{aligned} H(Y|X) &= E_p \ln \frac{1}{p(Y|X)} \\ &= -\sum_{i=1}^{n} \sum_{j=1}^{m} p(x_i, y_j) \ln p(y_j|x_i) \end{aligned} \tag{2.12}$$

If Y could be determined by X, *i.e.*, there is a function f such that $p(Y = f(X))$ approaches 1, then the conditional entropy $H(Y|X)$ approaches zero.

Mutual information (MI) measures the reduction in uncertainty of a random variable Y given another random variable X. This reduction represents the amount of information either variable provides about the other. It is defined by:

$$I(X, Y) = H(Y) - H(Y|X) \tag{2.13}$$

However, it is impractical to use either conditional entropy or MI as a measure of the similarity between X and Y. Conditional entropy is not symmetric, *i.e.*, $H(Y|X)$ is usually not equal to $H(X|Y)$. While MI is symmetric, its absolute value is not necessarily comparable across random variables. MI is influenced by $H(X)$ and $H(Y)$, which may have different maximal values. Strehl *et al.* [73] developed a normalization for MI, called Normalized Mutual Information (NMI), to address these shortcomings. It is defined by:

$$\text{NMI}(X,Y) = \frac{I(X,Y)}{\sqrt{H(X)H(Y)}} \tag{2.14}$$

For any random variable X and Y, NMI has the following nice properties:

1. $0 \leq \text{NMI}(X,Y) \leq 1$
2. $\text{NMI}(X,Y) = \text{NMI}(Y,X)$
3. If X and Y are independent, $\text{NMI}(X,Y) = 0$
4. If $Y = f(X)$, $\text{NMI}(X,Y) = \sqrt{\frac{H(Y)}{H(X)}} \leq 1$, for any function f
5. If $Y = f(X)$, $\text{NMI}(X,Y) = 1$, for any invertible function f

Consider two random variable P and Q with $H(P) = H(X)$ and $H(Q) = H(Y)$. We have

$$\text{NMI}(P,Q) = \frac{I(P,Q)}{\sqrt{H(P)H(Q)}} = \frac{I(P,Q)}{\sqrt{H(X)H(Y)}} \tag{2.15}$$

since

$$I(P,Q) = H(Q) - H(Q|P) = H(Y) - H(Q|P) \tag{2.16}$$

We have

$$\text{NMI}(P,Q) = \frac{H(Y) - H(Q|P)}{\sqrt{H(X)H(Y)}} = \sqrt{\frac{H(Y)}{H(X)}} - \frac{H(Q|P)}{\sqrt{H(X)H(Y)}} \tag{2.17}$$

Therefore, we can conclude that $\text{NMI}(P,Q)$ will increase as $H(Q|P)$ decreases. When $H(Q|P)$ is zero, $\text{NMI}(P,Q)$ will reach its maximum $\sqrt{\frac{H(Y)}{H(X)}}$. When $H(Q|P)$ reach its maximum $H(Q)$, $\text{NMI}(P,Q)$ will reach its minimum 0. In other word, the less uncertainty (measured by the entropy) one random variable have when the other random variable is known, the higher NMI of the two random variables are, given that the uncertainties of both variables are fixed. Therefore, NMI provides a good measure of the relationship between two variables, regardless of the specific form of the relationship.

Given the background information provided, in the next chapter we provide an overview of the prior research on monitoring complex software systems. Much of the

prior work has been applied to systems built using component-based frameworks such as Java EE.

Chapter 3

Related Work

Two observations are made when surveying related works in research to monitor the system by analyzing the metrics. First, people do not generally spend efforts differentiate the metrics according to their physical origins in the system. In other words, people do not involve system-specific knowledge when study the metrics. The reason is clear given the idea of autonomic computing: minimal system-specific knowledge is known so that the solution could be easily migrated to different environment or systems. Therefore, all metrics collected are considered equally informative until specific modeling techniques are applied, and the nature of the metrics are usually not taken into consideration.

The second observation is that efforts are usually made on modeling and studying the correlation between metrics, instead of on the metrics themselves. The reason is that the actual value of most metrics are neither stable nor following any specific or easily identifiable patterns. For example, The readings of metrics may simply fluctuate when workload changes, which is very common for most software systems. However, the workload is usually unpredictable, can fluctuate arbitrarily. For instance, the sudden increase of the CPU usage of an application server may be the normal response of increasing incoming requests. In other words, just focusing on the CPU usage itself does not directly provide much useful information on the healthy of the system. On the other hand, the relationship between metrics in the system may be much more stable and may reflect more interesting information of the system.

3.1 System Monitoring Based on Linear Regression Models

A large proportion of previous work are builds on the premise that a correctly functioning enterprise-software system exhibits long-term, stable correlations between many of its monitoring metrics [22, 34, 33, 62, 63, 64]. In addition, a large proportion of them focus on linear correlation between management metrics because such correlation is widely observed between system metrics and the modeling is simple and effective.

Diao *et al.* [17] have proposed a framework whereby multiple linear regression models are created for metrics of interest. They use the ordinary least square regression to establish models as

$$y = b_0 + b_1 x_1 + b_2 x_2 + \cdots + b_n x_n \tag{3.1}$$

for system modeling. However, they only focused on the finding of correlation models, and did not propose further steps for system monitoring and fault diagnosis.

Later, Munawar *et al.* [62, 63, 64] propose a full framework to quantitatively describe the underlying correlations between metrics and further use them to discover faults in the system. To identify stable linear correlations, Munawar collect metric data for a sufficiently long period to capture a representative behavior of the system. They then perform a pairwise correlation test on this data to find stable linear correlations, and further validate them using cross validations.

In the prior work [64] they rely on the *coefficient of determination*,R^2, to identify stable linear correlations. They use the Cook's Distance (Cook's D) [15] regression diagnostic to check whether predictions of the models hold for new samples. When the diagnostic value, which is computed using the model's prediction and the corresponding observed sample, exceeds a predefined threshold, the associated linear correlation is considered to have been violated.

Therefore, identification of stable linear correlations consists of two separate steps. First, they carry out experiments under normal condition, and collect samples of metrics. Then they pair-wisely study the samples, and retain those correlations with high R^2 as potential models. Second, they carry out more experiments and collect a few more samples and check if these new samples are explained with the models discovered. If more than a threshold portion of new samples are successfully explained, they consider the models stable.

Once the stable metric correlations are identified, the models can be used to detect errors and failures, and help in root-cause analysis by limiting the fault to a few components. To detect errors in the system, the status of all identified models is checked to determine how many report outliers. An error is suspected when a significant portion of all models report outliers, usually defined by another threshold.

The diagnosis is performed by assigning anomaly scores to metrics, ranking them, and then, retrieving components to which the metrics belong. It is assumed that the metric-to-component mapping is known. The more invariants associated with a metric that are violated, the higher the anomaly score. The diagnosis result short-lists a set of components which are deemed most likely to be faulty. The presumption of this approach is that faults often cause many metrics of the affected component to misbehave, which in turn make it rank high.

The problem with this solution is that it relies on many thresholds to work. Moreover, there is little guidance on how to set the right threshold. First of all, correlation candidates are identified by the threshold of R^2. In practice, people usually consider somewhere from 0.6 to 0.9 as the minimal R^2 for a pair to be considered linearly correlated. In [64], the threshold is set to 0.6 without any reasoning process. Second, to determine if a new sample fits a model, they need threshold for the diagnostic Cook's D, which is set to 2 without reasoning or explanation. However, these thresholds can still be justified by a lot of experience and relevant researches. The most critical threshold is the next one: to determine if an error should be suspected, the number of models should exceed a proportion of the total models. The proportion is another threshold, and is set to 0.5% in the evaluation. This threshold is hard to pick beforehand, and it is possible that the proper thresholds may be different for different applications. On the other hand, there is no sensitivity analysis for these thresholds so we do not know how critical these thresholds are.

3.2 System Monitoring Based on Other Linear Models

Jiang *et al.* [34] developed similar approaches to Munawar. While most procedures are similar, their mathematical tools are a bit different: instead of simple linear regression, they use auto-regressive regression with exogenous Input (ARX); they

develop a fitness score that is similar to R^2 to determine the fitness of their models. They use Jaccard coefficient to do diagnosis, based on the same idea that faults often cause many metrics of the affected component to misbehave.

The first problem with cross autoregressive models is that it is much more costly compared with simple linear models. In system monitoring, determining invariants over all metrics is $O(n^2)$ times the cost of building one model, where n is the number of metrics. Building an ARX model is much more costly compared with building a simple linear regression model. Therefore, Jiang *et al.* also make great efforts in developing approximate algorithms in terms of reducing the number of correlation tests [35]. Zhang *et al.* also proposed another approximate algorithm in [88].

The second problem is similar to that of Munawar's work. Jiang's work also relies on a number of thresholds to work. Moreover, they need to choose proper parameters just to establish the ARX model. The ARX model describes the following relationship between the input and output:

$$y(t) + a_1 y(t-1) + \cdots + a_n y(t-n) = b_0 x(t-k) + \cdots + b_m x(t-k-m) \quad (3.2)$$

where [n,m,k] is the order of the model and it determines how many previous steps are affecting the current output.

Therefore, there is a problem to choose the right [n,m,k] to establish the model. However, there is no evidence which order is suitable. Therefore, they set a range of the order [n; m; k] rather than a fixed number to learn a list of model candidates and then a right model is selected from them according to their performance in experiments. If the order range is set too small, the right model may not be included; on the other hand, if the order range is set too large, over fitting may become a problem and the computational cost will increase by multiple times. In [34], they use $0 \leq n, m, k \leq 2$ as the range of the order, which may be due to the limit of computational cost. The method is also suffered from the threshold-picking problem as the one for Munawar's solution.

3.3 System Monitoring Based on Non-linear Models

Non-linear modeling techniques (see *e.g.*, [22, 64]) are also studied in the hope that they may provide better metric coverage, and discover useful correlations which are in forms other than linear.

These works have very similar procedures as methods in section 3.1 and 3.2: collect metrics, apply a specific model, retain models with fitness score passing a threshold, test new samples with retained models and predict faults, diagnose based on anomaly scores which usually comes from variants of Jaccard coefficient. In chapter 5, we will abstract these procedures into four steps.

Munawar et al. [64] try Simple Linear Regression with transformed data in their work to deal with non-linear correlations. They use the model of the following form:

$$T(y) = b_0 + b_1 T'(x) \tag{3.3}$$

where $T(\dot{})$ and $T'(\dot{})$ are transformation used in analysis. It is, however, very critical to find the proper transformation beforehand. In their studies, they try a few simple functions like logarithm($T(x) = log(1 + x)$), inverse($T(x) = \frac{1}{1+x}$) and square root ($T(x) = \sqrt{x}$). Each type of models may describe a small portion of all non-linear relationships.

Another attempt made by Munawar et al. [64] is the use of Locally-Weighted Regression, which minimizes the locally weighted sum of squared residuals $\sum w_i^2(y_i - \hat{y}_i)^2$. The weight $w_i = K(\frac{D(x_i, x_{query})}{h})$, where $K(\dot{})$ is some weighting function, x_{query} is the independent variable, $D(\dot{})$ is the distance, and h is the kernel width set beforehand.

Gaussian Mixture Models is another modeling technique proposed [22] to model the non-linear relationship between metrics. They prefer this model because they have observed that data points for a few metrics pairs can be clustered together around several centers with compact cluster size. Therefore they apply the specific Gaussian Mixture Models to capture such behaviors:

$$p(z_i|\theta) = \sum_{j=1}^{G} \alpha_j p_j(z_i|\mu_j, \Sigma_j) \tag{3.4}$$

where the probability of observing data points $z_i = (x_i, y_i)$ is considered as the sum of observing them in G Gaussian-distributed clusters. μ_j, Σ_j are the parameters for multi-dimension Gaussian distribution, and α_j's are the unknown proportions of these mixtures.

In these paper the above models are found to be useful in detecting faults. However, most of them still suffer from a few problems to be practically efficient.

In the case of SLRT, for example, usually only a few functions like logarithm, inverse, and square functions are involved. In fact, most of these techniques are only modeling some specific form of relationships. In addition, these techniques usually require careful parametrization. For example, GMM and LWR both require appropriately setting critical parameters for the modeling to be effective. GMM requires finding the right number of clusters to model, and LWR requires choosing the right value for selecting the smoothing parameter. It is not clear if such parameter selection can be performed before the model is used or even if it will be robust.

Also, techniques such as GMM and LWR are computationally costly. Learning GMM is generally done using algorithms such as Expectation Maximization [86], with a cost of approximately $O(sck)$ for each model, where c is the number of clusters, s is the sample size used for learning, and k is the number of iterations required for convergence. For an LWR model, each prediction requires finding the nearest neighbors in order to fit a local regression. The cost of this is approximately $O(s^2 \log(s))$. Because a large number of such models may be identified for a system, their use would cause high overhead. Considering the limited type of correlation these models cover, the gain of using these models in system monitoring may be costly.

Bulej *et al.* [5] proposed clustering the recorded response times with the k-means clustering algorithm and then compare the performance between two tests. The accuracy of this method depends highly on the quality of the clusters generated.

Malik *et al.* [49] use Principal Component Analysis to discover clusters of counters that are correlated to each other. These clusters are also used to detects performance deviations among subsystems. In our view, this is also localizing faults in a new performance test that show anomalous behaviors.

3.4 Fault Diagnosis

Fault diagnosis has traditionally required that all managed entities, events they generate, and dependencies among entities to be specified in advance. For example, Yemini *et al.* use the information of the system structure to establish the causality graph to monitor and analyze systems [87]. Brown *et al.* [4, 24] in their work analyze correlation between metrics, and then propose procedures to infer the root cause giving the dependency graph manually. However, this is costly and often impractical. In general, the more information is required by an diagnosis system,

the less likely the system may be applied to a wide variety of systems. Even if these detailed system information is available, keeping it up-to-date itself would be challenging or at least costly.

Fault diagnosis is also possible when detailed negative symptoms can be observed and reported. For example, Tang *et al.* [82] develop an evidential overlay fault diagnosis framework to diagnose faults in overlay networks. They first identify a set of potential faulty components based on shared end-user observed negative symptoms, then dynamically constructs a plausible fault graph to locate the root causes of end-user observed negative symptoms. This work has the advantage that in a network environment, many negative symptoms could be observed and reported from individual sources, which is not present in our target systems.

More recent approaches to diagnosis leverage statistical and machine learning techniques to somehow automate the process. A few early attempt was made by Chen *et al.*, who point out that traditional problem determination techniques rely on static dependency models that are difficult to generate accurately in today's large, distributed, and dynamic application environments such as e-commerce systems [11]. Instead, they propose the solution by dynamically tracing real client requests through a system, and for each request they record its believed success or failure and the set of components used to service it. They then use Jaccard coefficients to measure the distance of components, and use a hierarchical clustering method to cluster components together. The distance of clusters they use in the method is the unweighted pair-wise arithmetic average of distance between components.

The method is shown to be working, and two years later, they keep the same idea but use decision trees as the new tool to process the information [10] and improves their work. However, they suffers from two drawbacks: first, tracing is expensive [56], and may add additional burden on the system; second, they have an orthogonal subsystem attempting to detect whether these client requests are successfully completing, which may not always be accurate. It is not evaluated how much burden it adds to the system, nor how reliable the orthogonal subsystem labels each request in their paper [10, 11].

Another work also use decision trees to analyze call path pattern to detect application-level failures. Their work is based on the assumption that the probability of certain components should be stable in call paths. Thus, they use chi-square test to determine if a call path is normal [45].

Our work, however, does not assume availability of trace-based path information

because the cost of collecting such information is prohibitively high. However, compared with error detection, fault diagnosis by studying metric correlations is more difficult. The use of correlation modeling has been shown to provide useful information for fault diagnosis [9, 33, 62, 64]. Jiang *et al.* [33, 62] proposed using Jaccard coefficient for diagnosis without evaluating its accuracy. The basic idea is to study the correlation between the models' states and the actual system state. If many models involving a component show anomaly when the system state is not normal, and many models involving that component do not show anomaly when the system state is normal, then the component will receive a high anomaly score.

In earlier work [62, 64] Munawar proposed a similar idea of calculating anomaly score; they assessed the accuracy of diagnosis with correlation models using fault injection in application components. In that work, they show that for many faults, they can have certain form of diagnosis, which consists of a ranked list of components likely to be faulty; in many cases they successfully have the faulty component in top 10 suspected components out of more than 30 components. However, it cannot pinpoint a single component as faulty or identify a specific fault.

3.4.1 Fault Diagnosis by Supervised Learning

The approaches discussed thus far are all unsupervised: only the normal behavior of the target system is learned beforehand. Therefore they are generally more available than the approaches with supervised learning, which requires data from faulty systems.

However, since it is common seen that the same problem may re-occur many times, and sometimes the faulty database is available, one other direction of fault diagnosis takes these advantage by applying pattern matching techniques to discover known faults. Once a fault has happened before and been solved, databases of knowledge of this fault may be established, and the fault may be recognized and diagnosed in the future. We do not attempt this type of fault diagnosis algorithms in this work because of the difficulty to classify different faults beforehand and to collect representative data with each type of fault. However, since this may be worth considered as a future work direction, we introduce a few representative works in this area in this section.

Brodie *et al.* show that a simple matching procedure works in detecting the most common problems by mapping call-stacks to faults, and resolving anomalous events based on knowledge from the database([3]).They generate failure symptoms

automatically with a simple algorithm, which simply try to find the longest identical piece of stacks among all stacks with each identified fault, with some adjustment for the location and relative length of the stacks. This is a very good try on this direction, however, the weakness is obvious: their approach is designed specifically for failure data that is in the form of program call stacks. As a result, their matching algorithm cannot be applied to other forms of data.

In a supervised approach, the system state needs to be classified *a priori* and then provided as input to the modeling technique (See *e.g.*, [11, 12]). Creating and keeping such knowledge base up-to-date is difficult. Despite the requirement, supervised learning can usually provide more accurate information in fault diagnosis for the faults that have been seen and studied. Not surprisingly, problems that have been resolved in the past should be detected and diagnosed more accurately and faster when encountered again. Cohen *et al.* [12, 13] propose an approach whereby Naive Bayesian models are used to correlate service-level objective (SLO) violations with low-level system metrics. However, the use of Naive Bayesian models in their work leads to the assumption that all metrics are independent given the system state, which is unlikely to be a good assumption. The system is complicated, and there are many dependencies and connections between its components. Therefore, it is unlikely that all metrics would be independent if the system is a normal state.

Recent attempts combine both the benefits of analytical models to describe metrics correlations and machine learning for recurrent faults [19, 36]. Both works first build a number of correlation models between metrics, and then use machine learning techniques to classify the system state into normal or with known faults. Ghanbari *et al.* [19] use Gaussian Mixture Models and Bayesian networks; Jiang *et al.* [36] use linear models and neural networks. They have different testbed therefore their evaluation reported in the two papers are not comparable. However, both pieces of work show that their ideas managed to help diagnosing recurrent faults.

Chapter 4

Problem Definition

We solve the problem of monitoring software systems in this thesis. More specifically, we work on the error detection and fault diagnosis in complex software systems. Before we give our definition of error detection and fault diagnosis we would like to study, we first outline the assumptions about what knowledge we know about the system, and what data is available during our monitoring. These information can be viewed as the input of the detection or diagnosis algorithms, and thus determines how well the algorithm could perform. Intuitively, the more information we have, the easier our algorithm could give accurate error information. On the other hand, some information are easier to access and some are harder to access. The more information the algorithm requires, the more difficult to collect these information and apply these algorithms to a wide range of software systems.

We divide information available to three groups according to the difficulty to obtain these information.

1. Level I: management metrics(defined in Section 2.2) collected from normal systems and monitored systems
2. Level II: structure of studied systems and/or relationship between system components and metrics
3. Level III: management metrics collected from systems with known faults present

The availability of level I information is a basic assumption of our work. Tools to collect management metrics is a reasonable assumption. Also, carrying out experiments to collect samples that are from a fault-free system is also easy as long as the target system can be running for a period which is believed to have no fault. Therefore our work assume that at a reasonable and acceptable cost, we are

provided with management metrics collected before we start our work.

The level II information is more difficult to collect. Since software is complex and dynamic, detailed system structure information may not be easily available. Sometimes even if the information is available, the system structure may frequently change so the information may be out of date easily. On the other hand, some high level system structure is readily available. For example, if a system consist of several machines connected by a network, then the system structure is clear in the machine level. If we assume the knowledge of the system structure, we usually also know the relationship between metrics and components. Such information is not enough for root-cause analysis for faults, but may enable fault localization, since we may have observed different metrics contributed differently to the disturbance of correlation existing in a system. Therefore we assume a high-level relationship map between metrics and components is provided when we talk about fault diagnosis in terms of fault localization.

The level III information is least available. For faults that never happened before, it is virtually impossible to access level III information. However, sometimes faults tend to reoccur, and for such faults some research labs may have database for past faults, therefore it may be available sometime. The benefit of availability of level III information is obvious: with level III information one can apply pattern matching techniques to quickly identify known faults. Therefore, level III information could potentially enable root-cause analysis of faults in a system. However, due to the difficulty to classify faults beforehand and the difficulty to collect the management metrics samples with real faults, we do not assume the knowledge of such information throughout this thesis.

After having assumptions of our knowledge of the system, we outline the system monitoring problem and the scope of our studies in Section 4.1 and Section 4.2.

4.1 Error Detection

Assume all metrics are indexed by the set $\{1, 2, \ldots, k\}$, the metrics sample we collected at any time t is stored in a vector with k dimensions: $\mathbf{m_t} = (m_{t1}, m_{t2}, ..., m_{tk})^T$. If we collect samples from time 1 to time T, we have T vectors and they form a matrix $\mathbf{M}$:

$$\mathbf{M} = \begin{pmatrix} m_{11} & m_{21} & \dots & m_{T1} \\ m_{12} & m_{22} & \dots & m_{T2} \\ m_{13} & m_{23} & \dots & m_{T3} \\ \dots & \dots & \dots & \dots \\ m_{1k} & m_{2k} & \dots & m_{Tk} \end{pmatrix}.$$

The system state can be given by a binary vector $S = (s_1, s_2, \dots, s_T)$ such that $s_t = 1$ if an error is present at time t and $s_t = 0$ otherwise. The S is correspond to M if both are for the same period of time in the same system.

If the system is in a health state from time 1 to time T, the S associated with M is simply $S = (0, 0, \dots, 0)^T$. M is our knowledge of normal behavior of metric samples. Another monitoring in a similar system with unknown system state from time $T' + 1$ to T'' yields another metrics matrix $\hat{M}$:

$$\hat{\mathbf{M}} = \begin{pmatrix} m_{(T'+1)1} & m_{(T'+2)1} & \dots & m_{T''1} \\ m_{(T'+1)2} & m_{(T'+2)2} & \dots & m_{T''2} \\ m_{(T'+1)3} & m_{(T'+2)3} & \dots & m_{T''3} \\ \dots & \dots & \dots & \dots \\ m_{(T'+1)k} & m_{(T'+2)k} & \dots & m_{T''k} \end{pmatrix}.$$

An error detection algorithm with only Level I information is an algorithm which takes only M and $\hat{M}$ as inputs, and output a binary vector $\hat{S} = (\hat{s}_{T'+1}, \hat{s}_{T'+2}, \dots, \hat{s}_{T''})$ such that $\hat{s}_t = 1$ if an error is claimed at time t and $\hat{s}_t = 0$ otherwise. The pre-assumed knowledge is that the S correspond to M is $S = (0, 0, \dots, 0)^T$.

A error detection algorithm with level III information is an algorithm which takes M, it's corresponding state vector S, and $\hat{M}$ as inputs, and output a binary vector $\hat{S} = (\hat{s}_{T'+1}, \hat{s}_{T'+2}, \dots, \hat{s}_{T''})$ such that $\hat{s}_t = 1$ if a fault is claimed at time t and $\hat{s}_t = 0$ otherwise.

Usually, level II information does not help error detection much. Therefore we make no assumption of error detection with level II information.

4.1.1 Measurement of Detection Quality

Determine if a system is in a normal state or in an anomaly state is a typical pattern recognition problem. Therefore, the quality of error detection is characterized by the recall and precision of the classification. We use the standard terminology as presented in Salfner *et al.*'s survey [70] throughout this thesis.

An ideal detection algorithm should report errors whenever there is an error and should not report an error otherwise. As such, our algorithm target the following:

1. *Recall:* when a fault occurs, the algorithm should report an error after the fault occurs and within a short detection window period.
2. *Precision:* any error reported should truly represent a fault in the system. Any error reported when there is no fault present is considered a false alarm and should be avoided.

Assume we apply some error detection algorithms many times to a system. If the error is present and an alarm is reported, we count it as a *true positive*. If the error is not present but an alarm is reported, it is counted as a *false positive*. If there is no alarm reported but there is an error present, it is counted as a *false negative*. If there is no alarm reported and there is no error present, it is counted as a *true negative*.

The recall is also known as the sensitivity, which is defined as:

$$Recall = \frac{true\,positive}{true\,positive + false\,negative}$$

The precision is defined as:

$$Precision = \frac{true\,positive}{true\,positive + false\,positive}$$

Sometimes we refer to its equivalence *False-positive Error Rate*, which is $1 - precision$ to measure the precision.

Sometimes it may be convenient to have a single measure to integrate the trade-off between recall and precision. The F-measure for this purpose, is defined as:

$$F - measure = \frac{2 \times precision \times recall}{precision + recall}$$

For any classification algorithm, there is always a balance between recall and precision. From our perspective, for an error detection algorithm we are willing to avoid any false positive, but may tolerate some false negatives. In another word, we want to make sure the false positive is very low, even if the price is to reduce fault coverage.

The reason of our preference is that each false alarm could have a high cost. When the system administrators are given an alarm, they must spend much effort

in checking the system, which could be very costly. If the false positive is high, the system administrator would rather not trust the alarms from the automated monitoring any more. On the other hand, even if some errors are missed by the automated monitoring but whenever an error is reported there is a true error in the system, the automated monitoring is still useful and trustable. An example is the email span-filter. If an email span-filter has a high false positive rate such that many important emails are classified as spams and are deleted, it would cause a lot of problems and people would rather not use such a spam-filter. On the other hand, if a span-filter never mark regular email as spam, but may reduce some true spam, it would still be valuable.

Therefore, we will try to develop an algorithm with low false positive, and try to improve the fault coverage when false positive is kept low.

4.2 Fault Diagnosis

Theoretically, different faults may influence the metrics and their correlations differently. Therefore, if Level III information is present, it is possible to match the observed metrics with previously recorded metrics to determine the root cause of the fault in the system. However, our solution presented in this thesis does not assume Level III information of the system. There are two major reasons for our decision. First, it is usually very difficult to access such information in practice. Second, it is also unlikely that all faults in the system have been seen before we create our diagnosis algorithm. As a result, an algorithm based on Level III information may not be able to address unseen faults. Therefore, there is always incentive to develop diagnosis algorithms without Level III information.

On the other hand, it is very difficult to develop algorithms to diagnose faults with only Level I information. Without knowledge of the metrics nor previous faults, there is virtually nothing we can do to diagnose any fault. Therefore, our work focus on fault diagnosis with Level I and Level II information.

The goal of our diagnosis is to help determine the source of faults. While detection answers the question of whether there exists a fault in the system, Diagnosis answers the question of “where the fault is in the system?”. The faster the source of a fault can be found, the faster its cause can be addressed. This reduces the amount of downtime the system incurs, thereby improving system availability. Therefore, we use the word “fault diagnosis” and “fault localization” interchangeably in this thesis, based on the scope of our work.

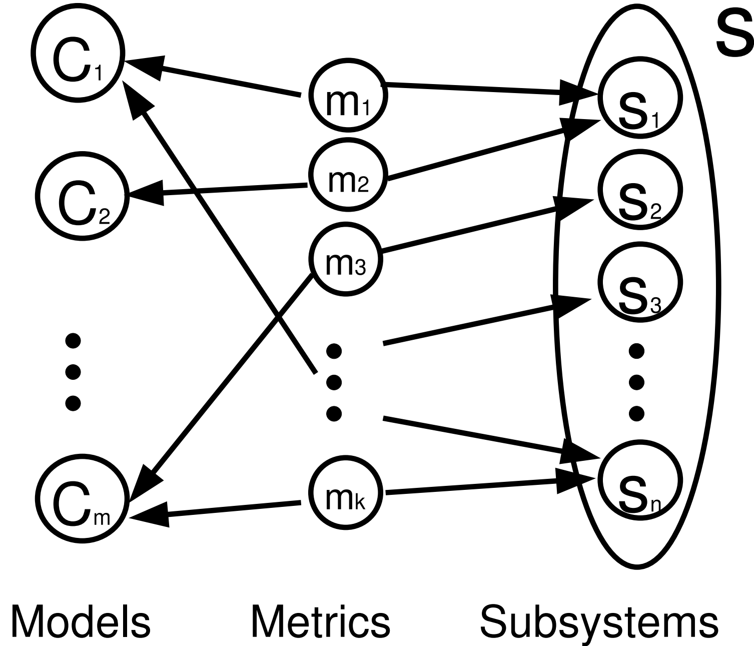


Figure 4.1: The relationship between models, metrics, and subsystems

We view the system as a collection of subsystems, and diagnosis localizes faults to a subset of the system. The smaller the subset is, the more precise the diagnosis is. The precision of diagnosis is determined by the monitoring data available, its collection cost, and the system administrators' needs. We consider any mathematical formula describing the relationship between some management metrics in the system as a model. These models forms the evidences for our diagnosis.

Fig 4.1 shows such a view of the system. A system $\mathbf{S}$ is a collection of subsystems, $S_1, S_2, ...S_n$. Let the set of all metrics be $\mathbf{M}$, and the set of all models be $\mathbf{C}$. Every metric $m \in \mathbf{M}$ of the system belongs to exactly one subsystem, $S \in \mathbf{S}$. On the other hand, a number of metrics can form a model $C \in \mathbf{C}$, and any metric may appear in multiple models. Therefore, there is a many-to-1 mapping from metrics to subsystems: $\alpha : \mathbf{M} \rightarrow \mathbf{S}$. Similarly, there is a many-to-1 mapping from metrics to models: $\beta : \mathbf{M} \rightarrow \mathbf{C}$. This implies that the relationship between models to subsystems is many-to-many, a simple mapping from models to subsystems may not exist. The relationship between models to subsystems can be modeled by a subset of the cardinality of models and subsystems: $\sigma \subset C \times S$. Therefore, identifying the faulty subsystems based on the anomalies in models is non-trivial.

Let $\mathbf{A}$ be the set of all mappings from subsystems to metrics, and $\mathbf{B}$ be the set of all mappings from models to metrics. By specifying the tuple $(\mathbf{S}, \mathbf{M}, \mathbf{C}, \mathbf{A}, \mathbf{B})$, we formally define the knowledge we assume about the system.

A model-subsystem association matrix [34] can be used to represent the relationship σ. The element of the matrix $M_{ij} = 1$ if there exist a metric $m \in \mathbf{M}$ and two mappings $\alpha \in \mathbf{A}$ and $\beta \in \mathbf{B}$ such that $\alpha(m) = S_i$ and $\beta(m) = C_j$, and $M_{ij} = 0$ otherwise.

At any time t, each model C_i is checked for anomalies. We record $o_i(t) = 1$ if C_i reports an anomaly, and $o_i(t) = 0$ otherwise. This gives us an observation vector $O(t) = [o_1(t), o_2(t), ...o_n(t)]^T$.

A diagnosis algorithm takes the matrix M for σ and the observation vector $O(t)$ as inputs, and outputs a anomaly score vector $\mathbf{r} = [r_1, r_2, ...r_n]^T$ such that the subsystem S_i is considered more likely faulty than S_j if $r_i > r_j$.

In our current work, which we discuss later, we have developed examples of diagnosis algorithms as defined here. In these algorithms, we consider software components as subsystems. Therefore, our diagnosis localizes faults at the software component level by integrating analysis results from all the models. However, with the same algorithms, we can change the tuple $(\mathbf{S}, \mathbf{M}, \mathbf{C}, \mathbf{A}, \mathbf{B})$ to extend our diagnosis to different models and different subsystem granularity.

Actually, different requirement of diagnosis may result in totally different level of difficulty and different ways to address the problems. For example, diagnosis in software component level is hard, and the resolution may require experienced software developers to look into the source code of the software. On the other hand, diagnosis in machine level may be much easier, and the resolution could be as easy as reboot a machine. Therefore, it is very important to carefully abstract the tuple $(\mathbf{S}, \mathbf{M}, \mathbf{C}, \mathbf{A}, \mathbf{B})$ according to the information available and resolution actions available in practice.

4.2.1 Measurement of Diagnosis Quality

The goal of diagnosis is to help determine the source of faults. The faster the source of a fault can be found, the faster its cause can be addressed. This reduces the amount of downtime the system incurs, thereby improving system availability.

In general, a good algorithm assigns the faulty component a higher anomaly score than other components. Hereby we define two measures of diagnosis accuracy,

the *Faulty Component Rank* and *Identified Fault Counts*. The identified fault counts could be computed given the faulty component rank so they are coherent. We use the faulty component rank to evaluate the quality of our diagnosis. However, the identified fault counts may be more practical for the system administrators who interpret the diagnosis.

Assume we applied a diagnosis algorithm to m cases where some fault is present and detected. We use $i = 1, 2, ..., m$ as indices of these cases. Let f_i be the number such that S_{f_i} is the faulty component in case i, and $\mathbf{r}_i$ be the anomaly score vector given by the diagnosis algorithm in case i.

Intuitively, the rank of the faulty component could be an indicator of the quality of the diagnosis. The faulty component rank R_i for the case i is given by:

$$R_i = \sum_{j=1}^{n} 1_{\mathbf{r}_{if_i} \leq \mathbf{r}_{ij}}$$

where $1_Q = 1$ if Q is true, and $1_Q = 0$ if Q is false. The smaller the R_i, the better the diagnosis is for the case i. $R_i = 1$ is the ideal case, which indicates that the diagnosis algorithm assign the faulty component the highest anomaly score. If the faulty component is assigned an anomaly score of 0, $R_i = n$, which is the worst case. Therefore, the vector $[R_1, R_2, ...R_m]$ is a measure of the quality of the diagnosis.

While intuitive, the faulty component rank alone is not very practical because the system administrators would not know the faulty component rank before they actually confirmed and identified the fault. To interpret the diagnosis, system administrators may consider components in order of decreasing anomaly scores. They first check the component with the highest anomaly score; if not faulty, they proceed to the one with the second highest anomaly score, and so on.

For practical reasons (*e.g.*, time availability), administrators may set a *candidate set size* t beforehand, and check if the faulty component is one of the components with t highest anomaly scores. If so, the fault is diagnosed, *i.e*, it is identified in the top-t components. For any t, the number of faults that are successfully identified in the top-t components (*i.e.*, the identified fault count) is given by:

$$N_t = \sum_{i=1}^{m} 1_{R_i \leq t}$$

Since there are only n components, $t \leq n$. t can be any integer value from 1 to

n chosen by the system administrator.

Chapter 5

Solution Framework

Many prior work on software system monitoring with management metrics modeling has similarities in their methods, regardless of the different correlation modeling techniques being used. We abstract the process in a solution framework in Section 5.1 and present our two solutions as two separate implementations of the solution framework in the Chapter 6 and Chapter 7.

5.1 A General Solution Framework for System Monitoring

Our solutions integrate many techniques to accomplish the task of software system monitoring. Hereby we give an overview of the solution framework. The framework is abstracted from many previous works on the similar problem and can accommodate many existing solutions.

A complete solution for software system monitoring with metrics modeling is consist of the following four steps:

- Metric Modeling
- System Health Signature Generating
- System State Checking
- Fault Localization

In **Metric Modeling**, assumptions for the behaviors of system metrics are made, and mathematical models are established to describe these assumed behaviors. Metrics collected from a health system is used to construct these models. Usually, the modeling is done offline so we can use costly complicated models.

In **System Health Signature Generating**, metric samples are collected during running of the target system and the monitoring is done by checking the collected metrics with the mathematical models established in metric modeling. Usually, every type of mathematical model has its own diagnostics, so the combination of all such diagnostics based on all mathematical models learned in metric modeling form the signature of the current health state of the system.

In **System State Checking**, a technique is used to estimate the system state: error present or not. This is usually done by a mapping from the system health signature to binary state estimations. Sensitivity and Accuracy is the key properties to evaluate the goodness of the system state checking.

In **Fault Localization**, we further use some techniques to turn the system health signature into localization of faults when an error state is found. This is usually done by a mapping from the system health signature to a faulty component in the system.

In general, most system monitoring based on metrics is consist of the four steps. By applying different techniques in any of the four steps, we can result in different system monitoring algorithms.

The techniques that may be used in each step is discussed in the following sections in this chapter.

5.2 Metric Modeling

Many models are proposed to describe the metric behaviors, and most of them assume there exist some relationships between metrics. They fall into two large categories: explicit correlations and implicit correlations. Specific models are used to describe explicit correlations, and other techniques are used to describe implicit correlations. A number of specific models that was used are introduced in this section.

5.2.1 Specific-Form Modeling

A number of modeling techniques have been proposed to characterize relationships between two metrics. The techniques proposed so far differ in terms of their explanation power as well as their computational cost for learning and tracking. In general, as the explanation power increases, so does the complexity and the cost of applying the technique.

These modeling techniques includes Simple Linear Regression, Simple Linear Regression with transformed data, Locally-Weighted Regression, Auto-regressive Regression with exogenous Input, and Gaussian Mixture Models, etc. (See Chapter 3).

Linear Models

Many specific models are proposed to model the correlation among management metrics. Linear regression models are the most well-established ones. Very frequently, there do exist a lot of linear correlations among management metrics. It is reported that there exist stable, long-term correlations among many metrics exposed by software systems.

Linear regression models usually have the following benefits:

- Linear correlations are widely observed across management metrics
- Linear regression models are cost efficient
- The confidence interval for new sample estimation is well-established, therefore ease our work in the second step "System Health Signature Generating".

The drawback of linear regression models are:

- Non-linear correlations are also observed between management metrics, therefore, using only linear regression models may lose information.
- To simplify the model, linear regression models make some assumption that may not be valid in real systems
- The confidence interval may be misleading if some of these assumptions are not met in real systems.

The other studied linear models is the auto-regressive regression with exogenous input (ARX). They were used by Jiang *el al.* [34] to model linear relationships. An ARX model predicts values based on past observations of the dependent variable and current as well as past observations of the independent variable. This is usually much more expensive than linear regression models in terms of computational cost. Moreover, they also suffer from the three drawbacks as linear regression models.

Non-linear Models

Examples of specific non-linear models studied thus far includes:

Simple Linear Regression with transformed data: This model is essentially the application SLR on smoothed data or data transformed using logarithm, inverse, power functions.

Locally-Weighted Regression: LWR is a non-linear regression technique that computes a local linear model for each prediction. As such, all points used as training are kept and used at query time. The local model is obtained by minimizing the locally weighted sum of squared residuals for all available points.

Gaussian Mixture Models: GMM captures the relationship between two metrics in the form of a set of Gaussians. GMM was used by Guo *et al.* [22] as well as Ghanbari and Amza [20] to track relationships between metrics.

Non-linear models usually have the following benefits:

- Some non-linear correlations are observed in the system. Therefore, non-linear models may be used to establish correlation models with more metric pairs.
- Use of non-linear correlations may improve the metrics coverage.

However, non-linear models have much more drawbacks:

- While many correlation could be "non-linear", each model can only model a few specific non-linear correlations
- Non-linear modeling techniques require careful parametrization. For example, GMM requires finding the right number of clusters to model, and LWR requires choosing the right value for selecting the smoothing parameter.

- most non-linear techniques such as GMM and LWR are computationally costly. Learning GMM is generally done using algorithms such as Expectation Maximization (EM) [86], with a cost of approximately $O(ksn^2)$, where n is the number of metrics, s is the sample size used for learning, and k is the number of iterations required for convergence. For LWR, each prediction requires finding the nearest neighbors in order to fit a local regression. The cost of this is approximately $O(rs\log(s))$ where r is number of retained models in LWR.

Therefore, it becomes a question whether it worth to improve system coverage by a little bit by introducing a specific form of costly non-linear model.

5.2.2 General-Form Modeling

In section 5.2.1 we introduce several existing modeling techniques to characterize relationships between metric pairs. However, all of them suffer from the same shortcoming: they assume an underlying mathematical form (*e.g.*, linear functions, non-linear functions, mixture of Gaussian distributions, *etc.*). However, there is no reason to believe that all relationships follow one specific mathematical form; therefore, each specific form must be modeled and computed separately, adding to the computation overhead, which makes their general application difficult.

An alternative way we propose is an information-theoretic approach to capture inter-metric relationship without the need to commit to any specific mathematical form for that relationship. Further, rather than pairwise comparison, we cluster similar metrics and monitor the resulting clusters, providing significant efficiency gains. The benefit of the general form models are as follows:

- We use an information-theoretic measure to quantify the strength of relationships between pairs of metrics. The measure derives from entropy and mutual information, and it can capture any relationship between metrics without assuming any specific form for the relationship.

- In contrast to prior work that entails modeling and tracking pairwise relationships between metrics, we group similar metrics together by employing clustering. We consider the resulting clusters as the entities that need to be tracked to monitor the health of the system.

- The number of models is much smaller compared with specific form modeling. For example, in the system we studied we usually end up with thousands of models with linear modeling when there is only a few hundred metrics because we have to model each pair of metrics separately. On the other hand, information-theoretic modeling usually end up with only a few models, which significantly improves the efficiency.

5.3 System-Health Signature Generation

Once we found a number of models that describe the behavior of the system in the metric modeling step, we can generate the system health signature of the system at each time period for any new sample from monitoring.

System health signature represents a collection of diagnostics of all models found in the metric modeling step. Assume n models are found during model learning process and indexed by 1,2,3,...n. For each new sample $\mathbf{m_t}$ at time t, the System Health Signature at time t is a vector of n dimension: $\mathbf{g_t} = \{g_{t1}, g_{t2}, ..., g_{tn}\}$, where g_{tk} represents the diagnostics by model k.

System health signature generating is a direct follow-up to the modeling we choose to implement. The signature used for the models depends on the type of models.

Examples of system health signature includes:

Binary Vector: Each specific form of direct models usually have its own way to determine if a new sample is an outlier. In this case, we can simply record $g_{tk} = 1$ if $\mathbf{m_t}$ is classified as an outlier by model k, and $g_{tk} = 0$ otherwise.

Outlier Count: When binary vector is used, the outlier count $w_t = \sum_{i=1}^{n} g_{ti}$ is usually used as a more condensed signature.

Cluster Entropy: In our work of the information-theoretic modeling, we group the metrics into n clusters. We make g_{tk} to be the in-cluster entropy of cluster k based on sample $\mathbf{m_t}$ and the resulted $\mathbf{g_t}$ is the system health signature.

5.4 System-State Checking

The system health signature generating gives us a time series of vectors $\mathbf{g_t}$, and the system state checking step produce the final classification vector $\hat{S} = (s_{T'+1}, s_{T'+2}, \ldots, s_{T''})$ as discussed in section 4.1.

Therefore, the system state checking is a mapping from $\mathbf{g_t}$ to s_t. Many different approaches are proposed to generate the function.

Many preliminary method use predetermined thresholds to make the decision. For example, many previous work use a specific form model in the metric modeling step, and outlier count in the system health signature generating step. The system state is determined by:

$$s_t = \begin{cases} 1, & w_t > \alpha n \\ 0, & otherwise \end{cases}$$

where $\alpha \in [0, 1]$ is a predetermined threshold. The idea is that if a large proportion of the models report outliers, the system is probably in an error state.

The major problem with such a method includes:

- It is very difficult to choose a proper threshold α. A lower α may lead to a lot of false positives, while a higher α could lead to a lot of high false negatives.
- For different systems, the best threshold α may vary. Therefore, applying the monitoring to different system may involve a lot of experimental efforts, which reduce the scalability of the monitoring solution.
- Even if a proper α could be found by experience for a specific system being studied, when there is a change in the metric modeling step, the proper threshold α could have to change. This will prevent the evolution of modeling techniques.
- Temporary disturbance ("spikes) may trigger a false alarm.

In this thesis we proposed a new way to do the system state checking with a non-parametric statistical test, namely the Wilcoxon Rank-Sum test [21], to identify significant shifts in the system signature. We have found the Wilcoxon Rank-Sum test to be most suitable for our needs. The Wilcoxon Rank-Sum test is relatively much better than a threshold-based function, as it does not learn a threshold or rely on one to work and it also allows temporary fluctuations to be accommodated.

The test is performed as follows. Let two sample sets be $X_1, X_2, ...X_n$ and $Y_1, Y_2, ...Y_m$, calculate the Wilcoxon Rank-Sum statistic:

$$W = \sum_{j=1}^{m} \sum_{i=1}^{n} h_{ij} + \frac{m(m+1)}{2} \tag{5.1}$$

where

$$h_{ij} = \begin{cases} 1, & X_i < Y_j \\ 0.5, & X_i = Y_j \\ 0, & otherwise \end{cases}$$

The computed statistic is compared to a critical value from the Wilcoxon Rank-Sum table to check whether the change is significant.

We detect errors as follows. Let X_i be the number of models that report outliers for sample i. In order to detect a significant change in X_i when an error occurs, we keep two sliding windows of X_i's. The test window consists of the most recent n X_i's. The baseline window consist of the m X_i's preceding the test window. We apply the Wilcoxon Rank-Sum test to the two windows. If the test indicates a shift between the two sets, an alarm is raised. Once alarms are raised, the baseline window is no longer updated to prevent adding anomalous observations.

5.5 Fault Localization

The fault localization is performed by assigning anomaly scores to system components, and then ranking the components by the anomaly scores. The fault localization short-lists a set of components which are most likely to be faulty.

Most current attempt to assign components anomaly score is based on the idea to count the number of times a component is found in anomalous models. The rationale for such algorithms are that a faulty component is likely to cause the models which involve the component's metrics to show anomalous behavior. As a result, a component has a higher anomaly score than other components if more clusters containing metrics of that component detect anomalies.

Given the model-subsystem association matrix M and the observation vector $O(t)$ as defined in section 4.2, a representative algorithm use the Jaccard coefficient to assign an anomaly score to each component:

$$r_j = \frac{\sum_{i=1}^{n} o_i(t) \cap M_{ij}}{\sum_{i=1}^{n} o_i(t) \cup M_{ij}}$$

Using an anomaly score based on the Jaccard coefficient is the most current diagnosis method based on metric correlations. Such methods was proposed in [34, 62] and evaluated in [62] in the context of metric-pair models.

Another way to do fault localization proposed so far involves level III information. If such additional information is available, then diagnosis may be made by pattern matching with know faults databases. For example, the pattern matching techniques may be artificial neural network or Bayesian network. In previous works it is reported that using such information and techniques may improve diagnosis accuracy. However, the availability of level III information is very difficult in practice and is not assumed in this thesis.

Chapter 6

Solution One: Linear Models

In this chapter we present our work in improving the linear regression model to monitor the system. Among all specific form models, we are particularly interested in linear regression models because it is the most cost efficient model, while it captures most frequently observed correlation, the linear correlation.

However, when we investigate and study system monitoring by linear regression models, we found out several factors that prevent linear regression models from effectively modeling the relationship between metrics even if the underlying relationship is indeed linear. We first discuss the several factors in section 6.1, and then present our solution to these problems in the rest of this chapter.

6.1 Problems of Simple Linear Regression

We encountered several problems when modeling linear correlated metrics. The most important one is that we observed significant presence of heteroscedasticity in the linear relationships.

6.1.1 Heteroscedasticity

In the statistics literature, *heteroscedasticity* refers to the fact that the variance of the residuals of a model is not constant. Heteroscedasticity is very commonly observed in applications of regression models. A popular example is the relationship between individuals' income and meal expenditure – there is greater variability in what an individual consumes as his/her income increases.

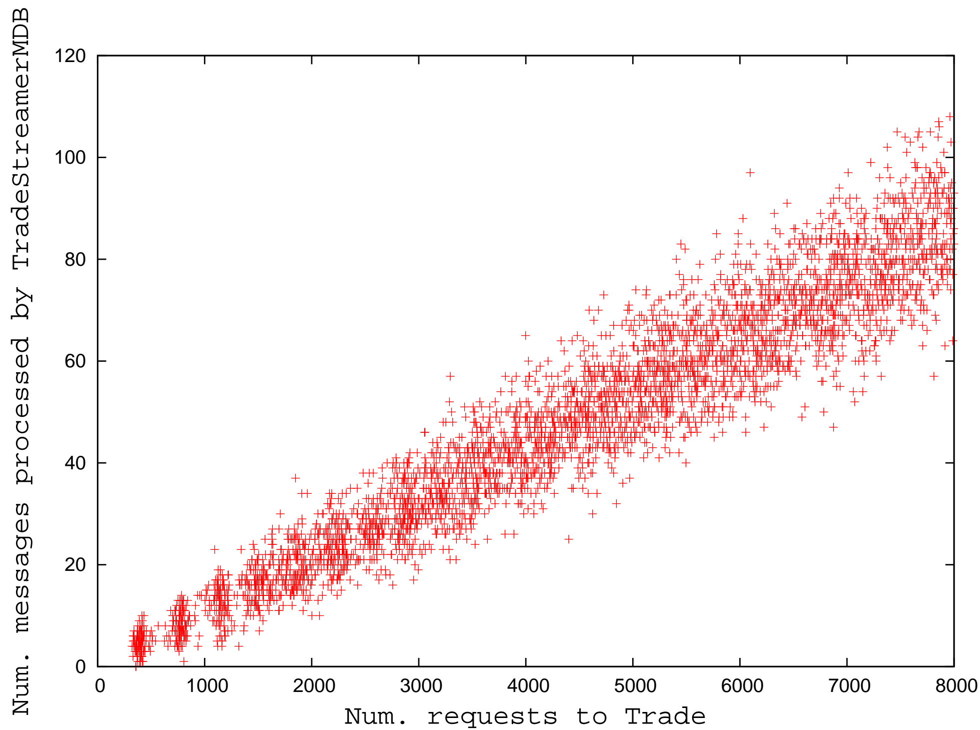


Figure 6.1: Heteroscedasticity example

When we study relationships between management metrics, we also observe such behavior in many instances (*i.e.*, the variance of the related variables is not constant over the observed range). An example is shown in Figure 6.1, which presents the scatter plot of two management metrics. We can see an overall linear relationship between the shown metrics. However, we also see that the variance of the two variables becomes larger as the values of the metrics increase, resulting in a covered area having a triangular shape.

The presence of heteroscedasticity in relationships between management metrics prevents us from monitoring these relationships effectively. In theory, such behavior violates the assumption of most regression techniques which stipulate that the error has constant variance (*i.e.*, the error covariance matrix is scalar). In the presence of heteroscedasticity, linear regression estimators based on Ordinary Least Squares may be biased and inconsistent. Moreover, heteroscedasticity biases the estimated standard errors, making many diagnostic measures unreliable. For example, the confidence intervals of model predictions become invalid. This is illustrated by the example shown in Figure 6.2.

Theoretically, a number of reasons may cause residuals of a regression model to display heteroscedasticity [68], including:

- *Varying regression coefficients*: As discussed in Section 6.1.2, while two met-

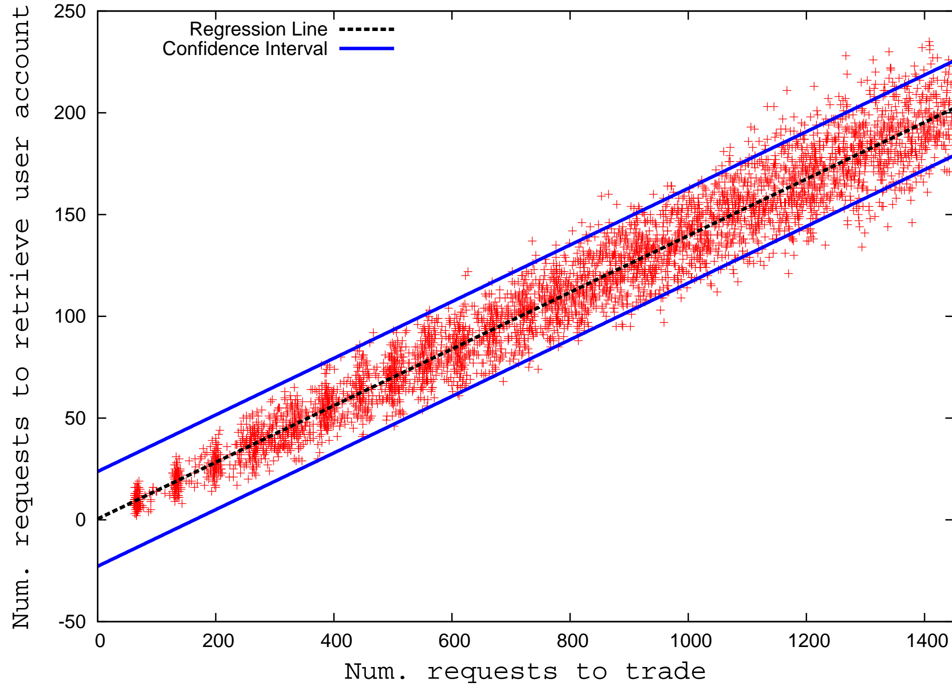


Figure 6.2: Confidence intervals using OLS regression

rics remain correlated, their regression coefficients may vary. If this behavior is not captured by the model, the residuals may include effects due to the variation of the coefficients. For example, if the true model is $y_i = \alpha + \beta_i x_i + e_i$, where the parameter β_i varies with i such that $\beta_i = \beta + \epsilon_i$, then our model becomes $y_i = \alpha + \beta x_i + (\epsilon_i x_i + e_i)$. The residuals $(\epsilon_i x_i + e_i)$ have a non-constant variance.

An example of such cases which we observed in our evaluation is given in section 6.1.2. The same pair of metrics are plotted in Figure 6.3 without differentiating between time intervals. Comparing with Figure 6.5, we can see that the non-constant residual variance is caused by changing regression coefficients, which vary with time.

- *Omitted variables*: As discussed in Section 6.1.3, models may not include a relevant variable, in which case the residuals will include the effects of that variable. This will cause residuals to vary with the missing variable. For example, if the true model is $y_i = \beta_0 + \beta_1 x_i + \beta_2 z_i + e_i$, but our model is of the form $y_i = \beta_0 + \beta_1 x_i + e'_i$, then the residual e'_i will include the effect of variable z_i. e'_i will not be normally distributed with constant variance because $e'_i = \beta_2 z_i + e_i$.

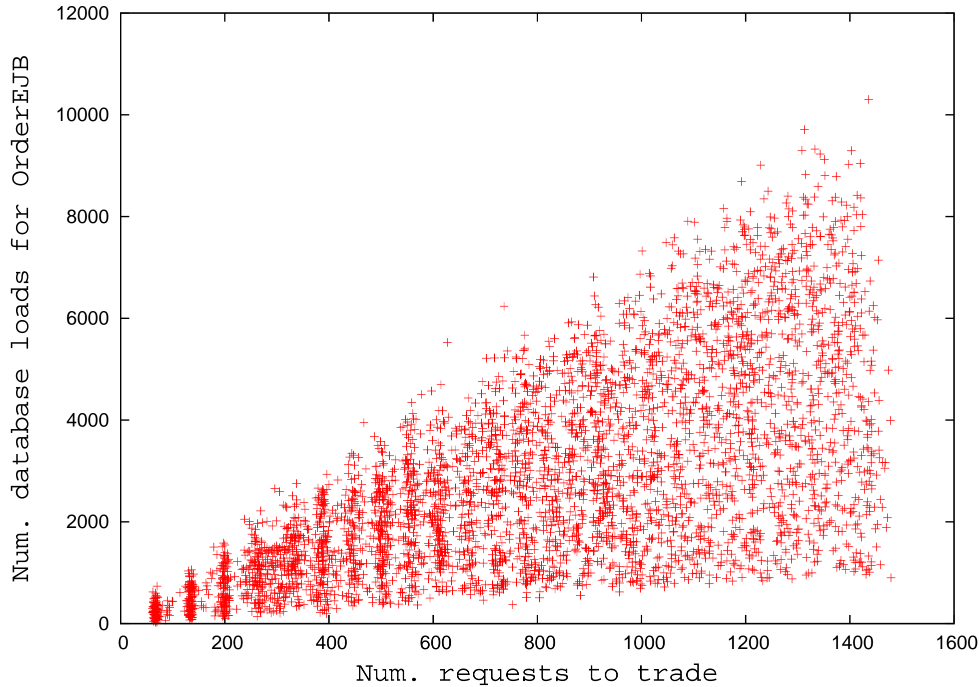


Figure 6.3: Heteroscedasticity example: varying coefficients

- *Inaccurate models*: If the model is not a good fit for the underlying relationship, then the residuals will capture the model inaccuracies. For example, if the true relationship is non-linear, but we capture it using a linear model, then the residual will exhibit heteroscedasticity. Non-linear relationships do exist in most software systems, especially in transaction-oriented systems [22, 37, 64].

In practice, we also observed several system-specific examples where the causes of heteroscedasticity could be identified, including:

- *Sampling errors*: Sometimes the metrics samples are aggregated (*e.g.*, using mean) and there exist groups in the sample data. As a result, the aggregate may be affected by the size of the groups. For example, when computing mean values, the smaller the group size, the more variance we will observe. This measurement error becomes part of the residuals, making them vary with the size of the underlying groups.

- *Caches and object pools:* At higher load, it is possible for caches and object pools to become full, creating variability in the response times and thereby the amount of work that can be completed. After the point where caches

and object pools become full, some metrics collected could show different behaviors. This may be modeled with segregated models or non-linear models. However, linear models is likely to produce residuals with non-constant variance.

- *Load-related variance:* As load increases, resource availability becomes more constrained. As such the unpredictability of system performance increases, leading to unpredictability of individual metrics that are performance-related. The variance of individual metrics contributes to the variance of residuals. This is an instance of omitted variables in the model. By taking into consideration the load-related effects, we may be able to capture the residual variance.

- *Application logic:* There could be application logic in the system that changes the metric relationships quantitatively. For example, in the system we study, the middleware on which our benchmarking application executes implements policies to allow the system to scale to increasing load. As the load increases, the number of threads available to handle work increases.

  A similar policy may also apply to object pools, whereby when load increases more objects are be pooled to better handle the larger volume of requests. As a result, varying coefficients of linear correlated metrics may be observed. If we do not capture these dynamics in the models, the effects will appear in the residuals.

- *Other system-specific reasons:* Another example of heteroscedasticity is shown in Figure 6.4, where the number of requests received by the application (X) is plotted against the number of requests meant to retrieve user account data (Y). This is a case where linear regression model does not accurately capture the underlying relationship. Usually, requests for different services (*e.g.*, browse, buy, update profile, *etc.*) provided by a transaction-oriented system follow some probability distribution. If the probability of a request for retrieving account data is p, then the number of requests to retrieve user account data (Y) out of X total requests follows a binomial distribution, $B(X, p)$. Therefore, the underlying relationship is $Y \backsim B(X, p)$. As such, the expectation of Y is pX, and the variance of Y is $p(1-p)X$. Therefore, the actual relationship is approximately linear but with variance that grows with the predictor.

In general, there are many factors, either general or system-specific, could lead to

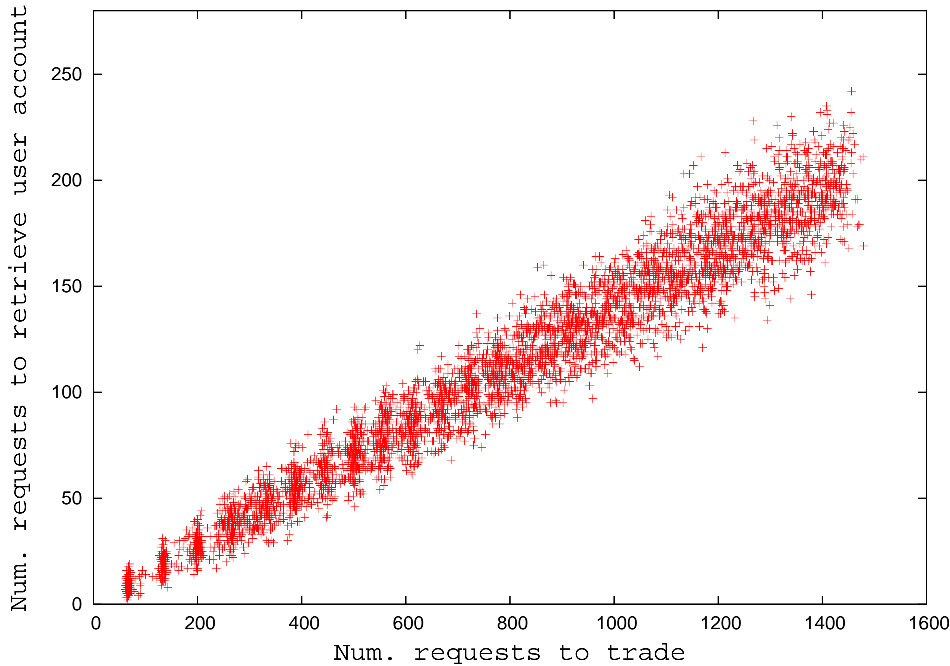


Figure 6.4: Heteroscedasticity example: inaccurate linear model

heteroscedasticity. Since heteroscedasticity is a common phenomenon, it is important to capture it when modeling metric relationships. Otherwise, the relationships we learn, especially predictions based on such relationships, may be misleading. To address this challenge, we need an approach to detect heteroscedasticity; we need specific models that capture the most frequent causes of heteroscedasticity such as varying coefficients and missing variables. However, since there are many possible causes of heteroscedasticity and there is no easy way to determine the true cause of heteroscedasticity, finding specific models to capture heteroscedasticity is not always feasible. Therefore, we also need a general approach to handle heteroscedasticity regardless of the underlying cause.

6.1.2 Varying Coefficients

One factor which prevents regression-based correlation models from correctly capturing system dynamics is that the model coefficients may change under different circumstances, even though the corresponding correlations still exist. For example, certain optimizations, either automatic or manual, may take effect to improve system performance; system operators may tune certain configuration parameters, or automatic optimizations may arise from shifts in the workload pattern. As discussed in Section 6.1.1, varying coefficients is also a common cause for heteroscedasticity.

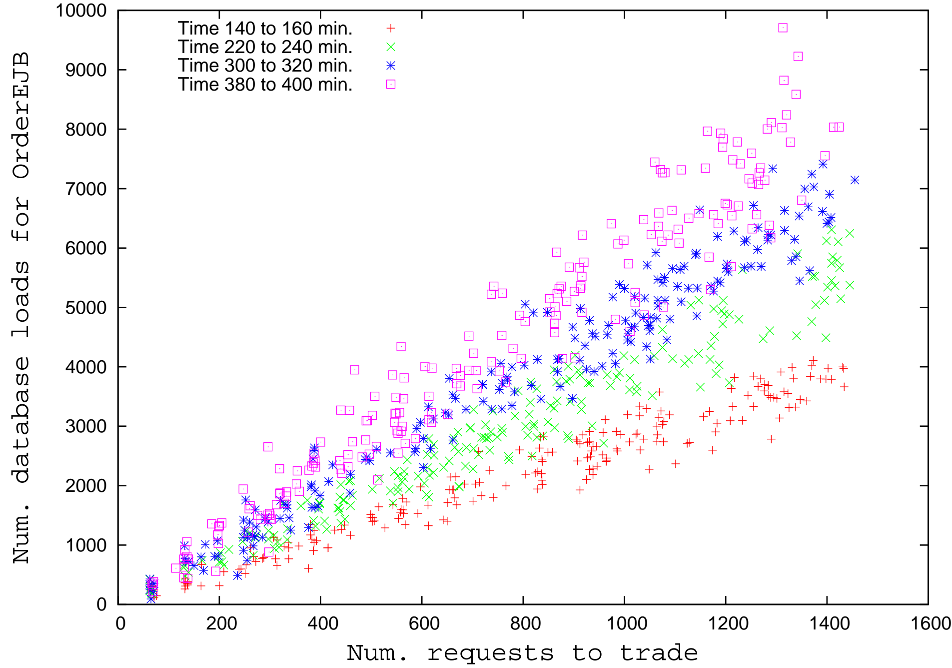


Figure 6.5: Varying coefficients of a metric-pair model

In Figure 6.5, we plot the samples of two correlated metrics collected during different time intervals with different colors. From this figure, we can see that the slope of the linear relationship is changing.

In this particular example, based on the testbed we describe later, the changing slope can be explained as follows: the relationship is between the number of requests received by the benchmarking application and number of order records that are loaded from the database. As time passes, the table grows in size as more orders are completed. Within any short time interval, the number of orders retrieved is proportional to the load, thus a linear correlation is observed. However, in distant time intervals, the number of orders retrieved per request (the slope) differs because more orders have completed and have been added to the order table.

In general, as systems evolve, many factors may influence the coefficients of correlation models. As a result, although the correlations still hold, models with out-of-date parameters may lead to inaccurate assessment of the modeled metrics. If we do not address the problem of varying coefficients, then the affected models may fail to hold in new circumstances without the presence of faults in the system.

There are generally two ways to address varying coefficients: 1) we can construct more powerful models to capture the varying coefficients; 2) because we may not always be able to include factors that cause coefficients to change, we need to

x =	Sample size	Models	F-score
24	617	$z = 24.25 - 4.32y$	237.15
25	90	$z = 25.44 - 4.47y$	39.89
26	312	$z = 26.64 - 5.01y$	226.77
28	981	$z = 27.93 - 3.85y$	276.00

Table 6.1: Regression coefficients varying with a third variable

be aware of the possibility of out-of-date models, and develop an error detection approach which can tolerate the failure of such models.

6.1.3 Multi-variable Correlations

A missing variable in a model may cause the coefficients to be unstable. An interesting example we have encountered is presented below.

There are three metrics involved in this example:

Metric x: `tradeEJB.AccountProfileEJB:LiveCount`

Metric y: `tradeEJB:MethodResponseTime`

Metric z: `tradeEJB.AccountProfileEJB:PooledCount`

The first metric, x, takes four values in our experiment: 24, 25, 26 and 28. If we collect samples of y and z according to different values of x, we find four linear models as shown in Table 6.1. We see that the constant term in these models varies with x, and including x would make the model more accurate.

The existence of multi-variable relationships has been observed in prior work [17]. Though, such relationships have been studied much less than those involving two variables. The reason is that the cost to iterate on every metric multiple times to test for possible multi-variable correlation is too high, while the gain from multi-variable correlation is not that significant.

In general, if there are n metrics in the system, and the cost for constructing a model with a group of selected variables is C, then searching for all two-variable correlations cost $O(n^2C)$ and searching for all k-variable correlations cost $O(n^kC)$. In addition, the number of multiple variable correlation could be as many as $O(n^k)$ which is the possible combinations of k metrics, while the number of two variable correlation should not be more than $O(n^2)$. Therefore, monitoring two variable correlation may be much more efficient. On the other hand, the majority strong relationships observed in most systems are still two-variable. Considering the sev-

eral magnitudes higher cost and the relatively less significant gain, we spent less efforts in developing multi-variable models for software systems.

6.2 Improving Simple Linear Regression

In this section, we propose our models to improve on two-variable Ordinary Least Square (OLS) linear models. Our first step is to detect non-constant error variance using well-established statistical tests.

6.2.1 Detecting Non-constant Error Variance

Many tests have been developed to test for heteroscedasticity, *e.g.*, White's General Heteroscedasticity test, Breusch-Pagan-Godfrey test and Goldfeld-Quandt test [16]. We choose to employ both the White's General Heteroscedasticity test and the Goldfeld-Quandt test in our work. The White test is the most general, regardless of the cause of heteroscedasticity. The Goldfeld-Quandt is more specific and supports the use of Generalized Least Squares (GLS) to model heteroscedasticity.

The White test for two-variable models consists of the following steps:

- Model the data (x, y) using ordinary regression:

  $$y = \beta_0 + \beta_1 x$$

  and obtain the residuals

  $$u = y - \hat{\beta}_0 - \hat{\beta}_1 x$$

- Regress u^2 against (x, x^2):

  $$u^2 = \gamma_0 + \gamma_1 x + \gamma_2 x^2$$

  Obtain R^2 of this regression.

- Compare nR^2 with the chi-square critical value $\chi^2_{\alpha,k}$, where n is the number of samples, α is the significance level of the statistical test, k is the number of regressors in the second step excluding the constant term, which is 2 in our case. If $nR^2 > \chi^2_{\alpha,k}$, heteroscedasticity is detected.

The White test is general. The Goldfeld-Quandt test, on the other hand, not only checks whether the error variance is constant, but also tests whether the variance is correlated with one of the independent variables in the model.

The Goldfeld-Quandt test for samples (x, y) involves the following steps:

- Order the observations according to the values of x, a variable to which the population error variance may be related.
- Omit c middle observations and divide the rest into the two groups of $(n-c)/2$ observations. The choice of c is arbitrary, but it is often chosen that $c = \frac{n}{3}$.
- Separately apply regression on the two groups by $y = \beta_0 + \beta_1 x$. Then, calculate the sum of residuals squared for the two groups, *i.e.*, SSE_1 and SSE_2:

$$SSE_1 = \sum_{i=1}^{(n-c)/2} (y_i - \hat{\beta}_0 - \hat{\beta}_1 x_i)^2$$

$$SSE_2 = \sum_{i=n-(n-c)/2}^{n} (y_i - \hat{\beta}_0 - \hat{\beta}_1 x_i)^2$$

- Compute the F-statistic thus:

$$F = \frac{SSE_2}{SSE_1}$$

- If $F > F_{\alpha,d,d}$, heteroscedasticity is detected, where $F_{\alpha,d,d}$ represents the critical value of F-distribution with significance level α, and degree of freedom d given by:

$$d = \frac{n - c - 2k - 2}{2}$$

  where k is the number of estimated coefficients excluding the constant term, which is 1 in our case.

6.2.2 Generalized Least Squares

A simple linear regression model is of the form:

$$\mathbf{y} = \mathbf{X}\beta + \mathbf{e}$$

where $\mathbf{e}$ is the random error whose variance matrix is given by $\mathbf{I}\sigma^2$, *i.e.*, the variance is uncorrelated, identically and independently normally distributed with fixed variance.

If the data does not pass the Goldfeld-Quandt test, the variance of the dependent variable gets larger when an independent variable gets larger. It is often assumed that the variance matrix of $\mathbf{e}$ is given by $\mathbf{C}\sigma^2$, where $\mathbf{C}$ is some known matrix,

while σ is unknown [16]. With this assumption, regression and estimation of new samples can be carried out as follows:

- Find matrix $\mathbf{P}$ such that $\mathbf{P}'\mathbf{P} = \mathbf{C}$. This results in a unique non-singular symmetric matrix.
- Let $\mathbf{y}^* = \mathbf{P^{-1}y}$, $\mathbf{X}^* = \mathbf{P^{-1}X}$, and $\mathbf{e}^* = \mathbf{P^{-1}e}$, we get the following new model:

$$\mathbf{y}^* = \mathbf{X}^*\beta + \mathbf{e}^*$$

  where $var(\mathbf{e}^*) = \mathbf{P^{-1}}\text{var}(\mathbf{e})\mathbf{P} = \mathbf{I}\sigma^2$.

- Applying OLS, the estimation of the parameter is given by:

$$\hat{\beta} = (\mathbf{X'C^{-1}X})^{-1}\mathbf{X'C^{-1}Y} \tag{6.1}$$

- An estimation of y, *i.e.*, $\hat{y}$, given any value of $\mathbf{x}$ is given by $\hat{y} = \mathbf{x}\hat{\beta}$.
- The confidence interval of y can be obtained by transforming the confidence interval of y^*. In OLS, the confidence interval is given by:

$$[\hat{y} - d(\mathbf{x})st_{(n-k-1),\alpha/2}, \hat{y} + d(\mathbf{x})st_{(n-k-1),\alpha/2}] \tag{6.2}$$

  where $k+1$ is the size of vector β, α is the desired significance level, and

$$s^2 = \frac{SSE}{n-k-1}$$

$$d^2(\mathbf{x}) = 1 + \frac{1}{n} + (\mathbf{x} - \bar{\mathbf{x}})'\mathbf{S}_{xx}^{-1}(\mathbf{x} - \bar{\mathbf{x}})$$

If the sample data passes the Goldfeld-Quandt test based on $\mathbf{X_i}$, the residual variance grows with the independent variable $\mathbf{X_i}$. Then, we can assume the residual variance matrix $\mathbf{C}$ is given by:

$$\mathbf{C} = \begin{pmatrix} x_{i1} & & & & \\ & x_{i2} & & & \\ & & x_{i3} & & \\ & & & \ddots & \\ & & & & x_{in} \end{pmatrix}.$$

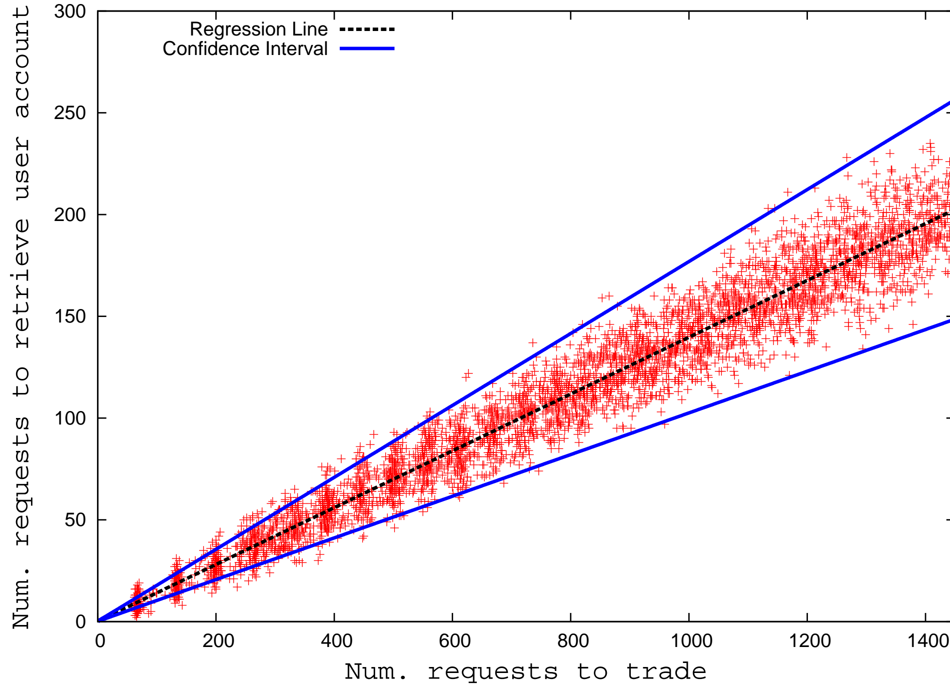


Figure 6.6: Confidence intervals using GLS regression

6.2.3 Fitness Score for Confidence Intervals

We use confidence intervals of predictions to estimate if a new sample fits the regression model learned; therefore, the better the confidence intervals approximates the actual distribution of the data samples, the better the model is. As such, we would prefer the confidence intervals in Figure 6.6 instead of those in Figure 6.2.

Let $U(x_0)$ be the upper bound of the confidence interval of y given independent variable x_0, and $L(x_0)$ be the lower bound. The area of the confidence interval for $x \in [a, b]$ can be given by:

$$\int_a^b U(x) - L(x)\, dx$$

Let $U'(x_0)$ be the empirical maximum value y takes when $x \approx x_0$, and $L'(x_0)$ be the empirical minimum value. The estimation of the area samples (x, y) with $x \in [a, b]$ occupies can be estimated by:

$$\int_a^b U'(x) - L'(x)\, dx$$

Therefore, The similarity between the two area can be estimated by the following

fitness score:

$$F = \frac{\int_a^b \max(\min(U(x), U'(x)) - \max(L(x), L'(x)), 0)\, dx}{\sqrt{\int_a^b U(x) - L(x)\, dx \int_a^b U'(x) - L'(x)\, dx}}$$

The fitness score ranges from 0 to 1. If the confidence intervals cover exactly the area which the observed samples occupy, the fitness score approximates its maximum 1. If the confidence intervals miss all the samples, the fitness score is at its minimum, *i.e.*, 0.

In practice, we take the $\alpha/2$ and $100-\alpha/2$ percentile of the interval all x's takes as a and b, and estimate the above integration.

For example, in Figure 6.6, suppose the grey triangle area is the area observed samples occupy. Ideal confidence interval should have $EFGH$ coincide with $ABDC$. Therefore, we use the following score to measure how well the model fits the observed data:

$$Fitscore = \frac{P(ABDC \bigcap EFGH)}{\sqrt{P(ABCD)P(EFGH)}} \tag{6.3}$$

where P(X) is the estimated area of X.

6.3 System-Monitoring Solution

We developed a new procedure for system monitoring in software systems according to the solution framework outlined in chapter 5.

1. *Metric Modeling*: We learn metric correlation models based on metric samples collected during a normal running period. This is usually done offline so the models are prepared before we start monitoring the target system.

2. *System-Health Signature Generation*: We use the outlier count w_t (defined in section 5.3) as the system-health signature. This signature is generated online as we collect real time samples from the monitored system.

3. *System State Checking*: We consider persistent changes in the system-health signature as an indication of existence of errors. Therefore, we use Wilcoxon Rank-Sum test to detect persistent changes in the outlier count for each type of models to do system state checking.

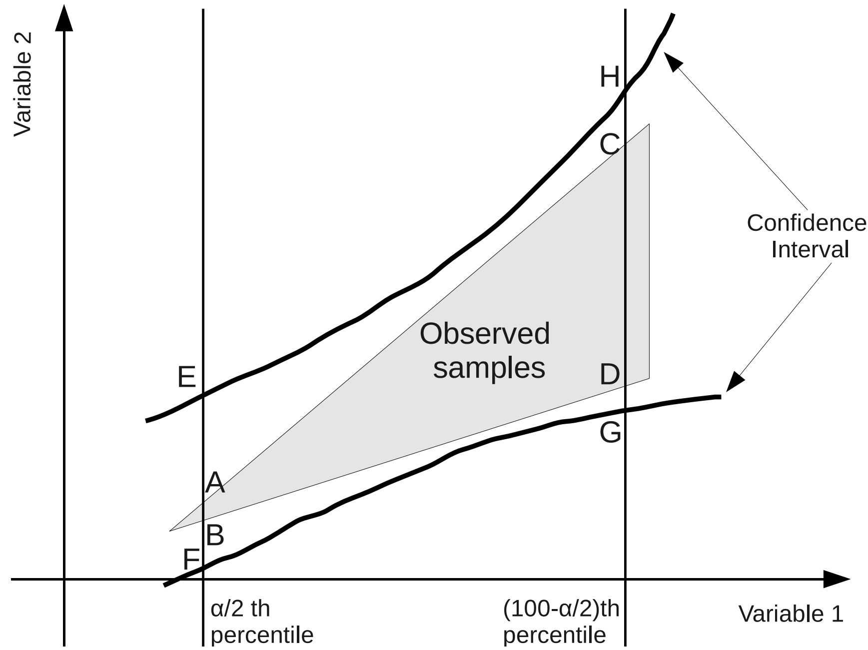


Figure 6.7: Fitness score calculation

4. *Fault Localization*: we use the Jaccard coefficient to assign anomaly scores to components and do fault localization based on the models we learned.

The approach is illustrated in in Fig 6.8 and the details of each step is discussed in the rest of this section.

6.3.1 Metric Modeling

We begin with model learning, which takes place offline based on metric data collected from a healthy system. Learning involves checking all pairwise combinations of metrics to identify strong correlations and estimating the corresponding regression models.

Figure 6.9 presents our approach to identifying the appropriate modeling technique. In the figure *pass* means that we accept the null hypothesis (*i.e.*, the error variance is constant). For each pair of metrics, we first employ the White test to see if the residual variance of two-variable linear regression models is constant. Next, the Goldfeld-Quandt test is used to check whether the GLS models can capture the observed heteroscedasticity. If both tests suggest that the residual variance is

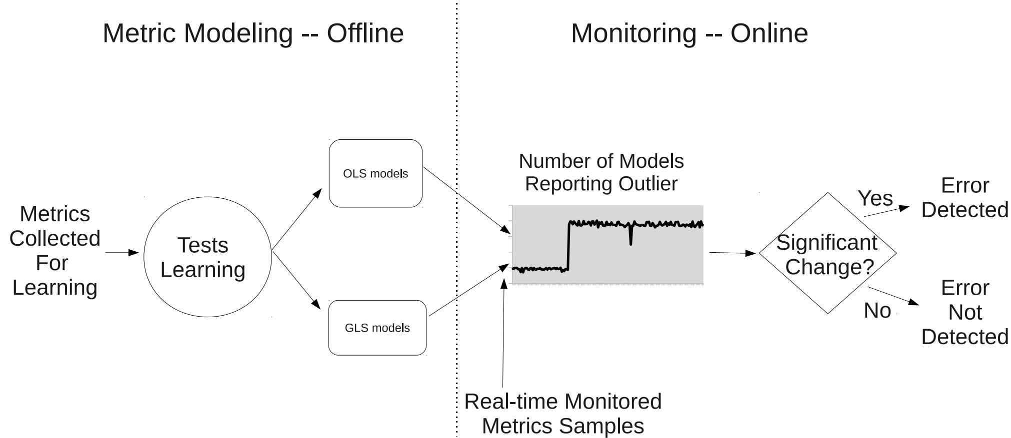


Figure 6.8: Model learning and system monitoring

constant, we model the relationship using OLS regression. If both the White and the Goldfeld-Quandt tests fail, which suggests that the use of GLS regression may be appropriate, we employ the GLS model. This procedure gives us two categories of models: OLS and GLS models. The model learning are done offline.

6.3.2 System-Health Signature Generation

First of all, we use confidence intervals in Equation 6.2 to estimate the acceptable range of the dependent variable for each model. If the observed value lies outside the confidence intervals, an outlier is reported by the model.

For each new sample we collected from real-time monitoring, we using all models we learned to test for outliers and aggregate the number of outliers reported by these models. In other word, we use the outlier counts w_t (defined in section 5.3) of all models as the system-health signature.

6.3.3 System-State Checking

In order to check the system state based on the outlier counts, we need to understand how outlier counts are expected to behave when the system is in a normal state or in an error state.

Theoretically, if there is no error in the system, the correlation between metrics are stable so most models we learned should not classify the new samples collected

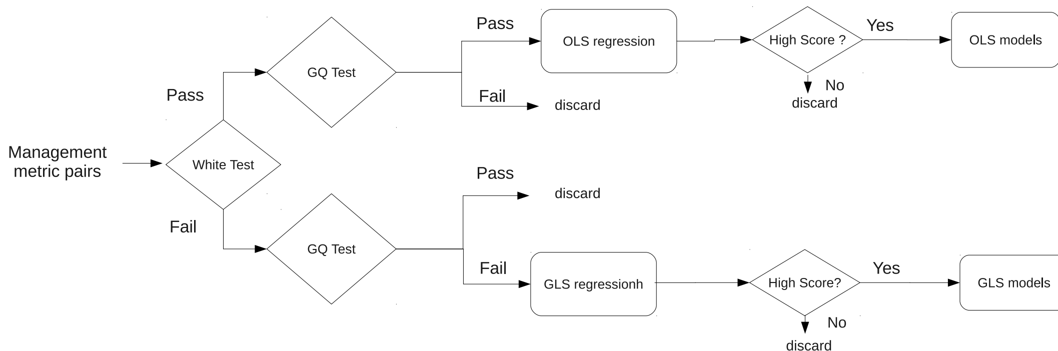


Figure 6.9: Learning metric-correlation models

from monitoring as an outlier. As a result, the outlier count should be relatively small compared with the number of models. However, the real system behaviors are usually more complicated.

Two major concerns need to be addressed when we do system state checking. First, there could be temporary "spikes", or simply a lot of false positives in the outlier counts, because of a some other temporary factors or just by a small probability. Therefore, an alarm should only be raised after investigating a few samples instead of a single sample. Second, coefficients we learned may be out of date because an environmental change in the monitoring period. Among the four categories of models we used, only one category of our models, the Recursive Least Square models account for varying coefficients; however, even when varying coefficients are explicitly modeled, it is still possible for some models to become invalid because of factors that were not considered at the time of modeling. As a result, we need to cater for cases where some of the models we learned become invalid when circumstances change. In other word, we need a detection mechanism that can tolerate failure of some of the learned models.

Given these considerations, we take a change in the outlier count as an indication of the existence of a error, rather than the value of a single outlier count. The rational is as follows: when the system is in a normal state, the models that are not valid report outliers. When the system has errors, additional models that were valid under normal conditions may also report outliers. Therefore, it is possible to see a change in the total number of models that report outliers.

Therefore we need a way to detect persistent shifts in number of the outlier

counts, as well as to tolerate the outliers reported by models with out-of-date coefficients. For this purpose, we use the Wilcoxon Rank-Sum test to detect such changes. The Wilcoxon Rank-Sum test is a non-parametric test; we thus do not need to specify a specific threshold on the number of failed models to indicate an anomaly. Also, if a portion of models report outliers from when the monitoring started, they will not be detected by the test.

By implementing the Wilcoxon Rank-Sum test to check the status of the system, we assume a relative and persistent change in the system health signature is the signal of error. Since we may have multiple categories of models, the Wilcoxon Rank-Sum test is applied to each category of models separately. If test on any one of these categories of models reports an anomaly, we raise an alarm. This approach is illustrated in in Fig 6.8.

The test is done as follows: assume w_t to be the outlier count for a category of models at time t. We keep two sliding windows of w_t's. The test window consists of the most recent n w_t's. The baseline window consists of the m w_t's preceding the test window. We apply the Wilcoxon Rank-Sum test to the two windows to determine if there is a persistent shift between values in the two windows.

The Wilcoxon Rank-Sum test is a well-established hypothesis test. In our case, the null hypothesis is that the two sample sets $\{w_{t+1}, w_{t+2}, ...w_{t+m}\}$ and $\{w_{t+m+1}, w_{t+m+2}, ...w_{t+n}\}$ from the two sample windows $(t+1, t+m)$ and $(t+m+1, t+n)$ are from the same distribution. The Wilcoxon Rank-Sum statistic is given by:

$$W = \sum_{j=1}^{m} \sum_{i=1}^{n} h_{t+i,t+m+j} + \frac{m(m+1)}{2}$$

where

$$h_{ij} = \begin{cases} 1, & w_i < w_j \\ 0.5, & w_i = w_j \\ 0, & otherwise \end{cases}$$

The computed statistic is compared to a critical value from the Wilcoxon Rank-Sum table to check whether the change is significant. If the null hypothesis is rejected, an anomaly is reported. Once an anomaly is reported, the baseline window is no longer updated to prevent adding anomalous observations to the baseline window.

6.3.4 Fault Localization

We can use the Jaccard coefficient to assign anomaly scores to components and do fault localization based on the models we learned.

Assume we have the model-subsystem association matrix M and the observation vector $O(t)$ as defined in section 4.2, the anomaly score r_j of component j is given by:

$$r_j = \frac{\sum_{i=1}^{n} o_i(t) \cap M_{ij}}{\sum_{i=1}^{n} o_i(t) \cup M_{ij}}$$

The rationale for this solution is that a faulty component is likely to cause the metrics to behave unusually. As a result, models which involve the component's metrics are more likely to show anomalous behavior. This is reasonable, however, we also notice that there are intrinsic limitations on the fault localization based on observing the correlations disturbed when a fault is present in the system. We discussed these observations in Appendix B.

Chapter 7

Solution Two: Information-Theoretic Models

In this chapter we present our second solution to the problem studied. We model general relationships by introducing the entropy concept from information theory, which effectively captures many non-linear correlations, and take the clustered pattern into account. This solution can capture more correlations and it is much more efficient.

7.1 Approach Overview

We propose an approach to monitor the health of a system as illustrated in Figure 7.1. The three steps are as follows:

1. We compute the NMI between metrics, which is a similarity measure independent of the underlying relationships.
2. We apply clustering, with NMI as the similarity measure, to group similar metrics together. Each cluster is a model of the correlations in the system.
3. We generate the system health signature by calculating the in-cluster entropy of each cluster. Then we check the state of the system by identifying persistent changes in the signature.

The above approach fits into our proposed solution framework. The first two steps are the metric modeling, which build up metric models by clustering similar metrics. The third step contains the system health signature generating and the

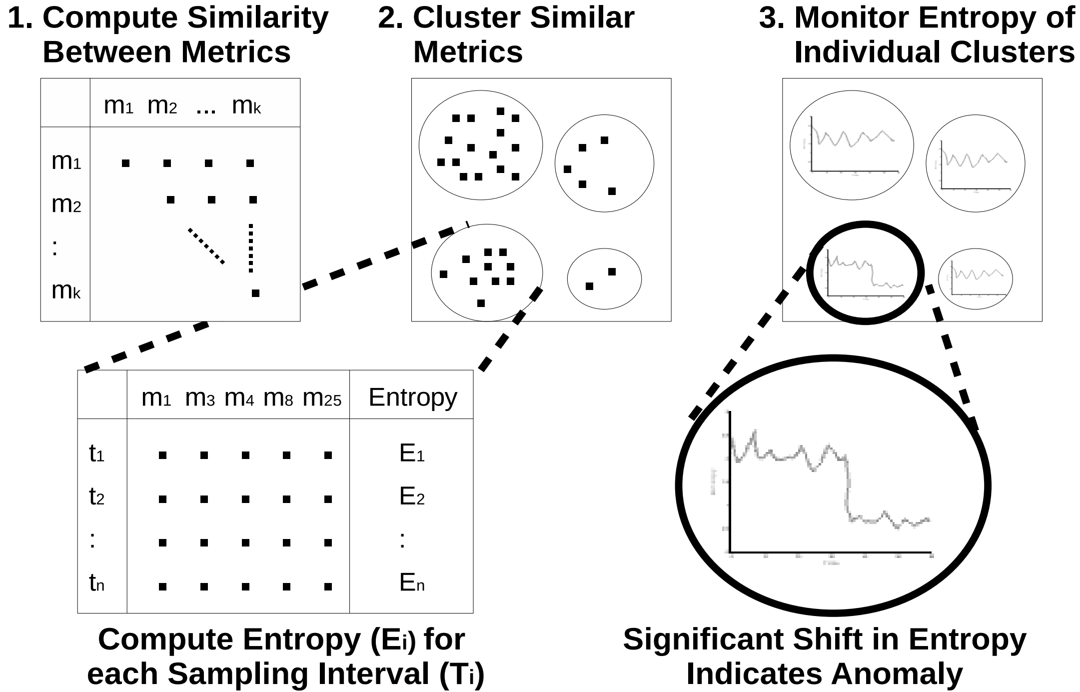


Figure 7.1: Approach overview

system state checking, where the in-cluster entropy of each cluster constitutes the system health signature and the system state checking is done by identifying persistent changes in the signature. The details of each step is discussed in the rest of this chapter.

7.2 Computing Similarities between Metrics

The information entropy measures the uncertainty or unpredictability of a random variable (see Section 2.4.2). The mutual information measures the uncertainty decreased for a variable when another variable is known, therefore measures the similarity between two variables. Therefore, we adopt the normalized mutual information as the measurement of similarity between metrics.

The theoretic value of the information entropy of any system metric is not known as we can only obtain a limited number of samples of that metric. Therefore, we use empirical entropy based on observed samples to estimate the information entropy and calculate the NMI. Computing the empirical entropy requires samples of the metrics collected. Therefore, we assume the availability of metrics periodically

collected from the target system over a period of time during which the system operates fault-free.

Given n observed samples of any metrics X, the empirical entropy $H(X)$ is then computed as follows:

$$H(X) = -\sum_{i=1}^{k} \frac{n_i}{n} \log \frac{n_i}{n}$$

where the observed n samples are divided into k bins and n_i is the number of samples observed in bin i. The empirical conditional entropy $H(Y|X)$ based on observed samples is computed as follows:

$$H(Y|X) = -\sum_{i} \sum_{j} \frac{n_{ij}}{n} \log \frac{n_{ij}}{n_i}$$

where n_{ij} is the number of samples (x, y) with x in bin i and y in bin j.

With the empirical entropy and empirical conditional entropy, we compute the normalized mutual information using Equations (2.13) and (2.14) for all pairs of metrics. This allows us to create a metrics similarity matrix as shown in step 1 of Figure 7.1.

7.3 Metric Modeling by Clustering Correlated Metrics

We start our modeling by observing the fact that if both (X,Y) and (Y,Z) have relationships, then (X,Z) often also have a relationship. For example, if $Y = f(X)$ and $Z = g(Y)$ deterministically, then $Z = g(f(X))$ hold. The relationship between management metrics may not always be deterministic, however, it is often observed in practice many such relationships cluster. In other word, there exist groups of mutually correlated metrics, or *clusters*. Consider a cluster of n metrics. If we model them by pairwise correlation models between every two metrics, this will end up with $O(n^2)$ models. On the other hand, if we model the cluster as a whole, we need only one model. This could provide a significant efficiency gain for the metric modeling. Therefore, we leverage clusters to improve the efficiency of tracking metrics. According to our four-step framework, we model the system metrics as a few clusters each of which consists of correlated metrics.

Given a similarity matrix as shown in step 1 of Figure 7.1, we apply the complete-link hierarchical agglomerative clustering (HAC) [23] to group similar

	High NMI	Low NMI
High r^2	Linearly correlated	Not possible
Low r^2	Non-linearly correlated	Not correlated

Table 7.1: Correlations captured by r^2 and NMI

metrics together. The algorithm takes a similarity matrixas input. The distance between two clusters is defined as the maximum distance between elements of the two clusters. It treats each metric as a single cluster, and then successively merges nearest clusters until either the distances between every two clusters exceed a predefined threshold, or all metrics belong to one cluster. The algorithm ensures that all metrics in a cluster have a similarity of at least t_{NMI}, a threshold that we specify. Therefore, we ensure that the system metrics are divided into clusters, such that all metrics within a cluster are correlated to each other.

7.3.1 Identifying Correlated Metrics

Given the empirical entropy and the normalized mutual information, we need a suitable threshold for the empirical entropy to differentiate weak correlations from strong correlations. In other word, we need to specify a value for the parameter t_{NMI} in our clustering algorithm. However, NMI is a relatively new and there is little guidance as to what value of NMI constitutes an indication of strong relationship. We resolve this issue by comparing it with the square of Pearson's correlation coefficient, r^2. r^2 has the same properties as the first three for NMI(see section 2.4.2); however, the fourth property holds only if f is a linear function. r^2 is a well-studied linear similarity measure, which is widely used.

Our work starts with the observation that any pair of metrics with strong linear correlation must have a strong correlation, but the reverse is not necessarily true. Strong non-linearly correlated pairs may have a low r^2 but a high NMI. This observation is summarized in Table 7.1.

Therefore, a suitable NMI threshold for strong correlation must not classify any pair of metrics with high r^2 as weakly correlated. Therefore, the proper NMI threshold t_{NMI} for strong correlation, with respect to the r^2 threshold for strong linear correlation t_{r^2}, is defined as:

$$t_{\text{NMI}} = \min_{r^2_{XY} > t_{r^2}} \text{NMI}(X, Y)$$

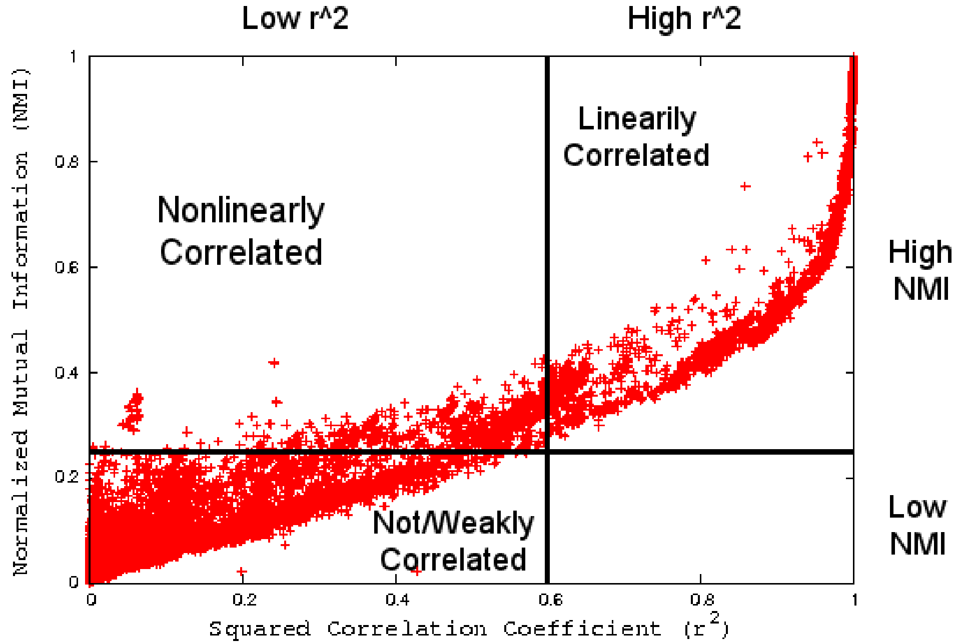


Figure 7.2: Metric similarity measures: r^2 and NMI

If we compute both r^2 and NMI for many metric pairs, we can have an estimation of t_{NMI} with respect to any t_{r^2}. This procedure is illustrated in Figure 7.2, which contains sample points from our experimental data(see section 8.3.1). Based on this figure, an NMI threshold from 0.25 to 0.5 seems proper. We also evaluate if our algorithm is sensitive to the NMI threshold in our evaluation(see section 8.3.4).

7.4 Tracking Groups of Related Metrics

According to the solution framework, we need to generate a signature for each cluster we have obtained. The metrics in one cluster are closely correlated to each other. However, since a relationship may not fit any pre-assumed specific form, it is non-trivial to establish analytical models for either metrics in the cluster or the cluster itself.

Information entropy, however, provides us with a tool to monitor the state of each cluster. Because metrics in one cluster are highly correlated, the uncertainty among values of the metrics in the cluster at a given time must be significantly lower than the uncertainty of the same number of uncorrelated metrics, regardless of the actual values of the metrics. Therefore, empirical entropy, which measures uncertainty, can be considered as a signature of the cluster, providing an indication

of the current status of the cluster. Therefore, the system health signature is a collection of cluster entropies of these clusters.

The procedure of computing in-cluster entropy for each cluster is as follows:

- Normalize all metrics values: although highly correlated, some metrics could range from thousands to millions, while others may never exceed one. We normalize each metric by dividing it by its average value, which is computed based on data collected during normal operation.
- We consider values of different metrics in a cluster as pertaining to a single random variable. Let W_t be the random variable for the studied cluster at time t, the value of metrics $m_1, \ldots, m_k$ in the cluster are considered as instances of W_t.
- At each time t when a sample is collected for all metrics, we calculate the empirical entropy of W_t, $E(W_t)$. We use static binning for this calculation; we set the range to $[0, 7]$ and divide it into 7 equal bins. We add an eighth bin with range $[7, \infty]$ to cater for values that do not fit in the other 7 bins. Our normalization entails dividing by the average; as such, we expect most data to lie within 7 factors of the mean.

As samples of metrics are collected at time t, we calculate $E(W_t)$ for all studied clusters.

The absolute values of entropy for different clusters are not comparable, as this value is affected by factors such as the size of the cluster and how data was binned. Nevertheless, the entropy of the same cluster at different times is comparable. For a given cluster, a significant change in the behavior of in-cluster entropy indicates a potential fault. This implicitly indicates that correlations among metrics are either disturbed or strengthened, both of which could be signs of anomalies.

Therefore, monitoring consists of tracking the entropy of each cluster. For each cluster, we monitor the in-cluster entropy $E(W_t)$ over time. We expect to see abrupt changes in the entropy behavior of some clusters when faults occur. This strategy is based on the observation of in-cluster entropy behaviors illustrated in Section 7.4.1. The detail procedure is discussed in Section 7.4.2.

The benefits of tracking correlations within a cluster by entropy includes robustness and efficiency. Small variations will not affect the entropy, as metrics with minor variations will tend to stay in their current bins and the empirical probability

will not be affected. Varying environmental factors such as changing workload are also unlikely to disturb entropy, as entropy does not depend on actual values but the uncertainty in the relative values of the metrics. If all metrics are mapped into some other bins separately, it is likely that the in-cluster entropy would not change significantly.

The cost of computing entropy for a cluster of size m is only $O(m)$; therefore, monitoring a system with n metrics, each pertaining to at most one cluster, only costs $O(n)$. On the other hand, checking all pair-wise correlations is $O(n^2)$ for a system with n modeled metrics, which is a much larger computation overhead.

7.4.1 Observations on Cluster Entropy

Consider the entropy behavior of the clusters shown in Figure 8.11, 8.12, 8.13, 8.14 and 8.15. They show the behavior of in-cluster entropy in some realistic experiments. Some fault occurs at time-sample 56 in all five cases.

Human operators can readily identify unusual changes in the level of the in-cluster entropy, and, as a result, suspect errors. However, it is impractical to have these operators continuously track the behavior of all clusters. On the other hand, automatically identifying anomalies in the in-cluster entropy is non-trivial because there are no general rules that differentiate between normal and disturbed behavior.

We considered several characteristics of the in-cluster entropy before devising a method to automatically identify anomalous behavior.

First, the empirical entropy estimated for different clusters are not comparable. As a result, no single threshold is suitable for all clusters. Thus, techniques based on setting thresholds do not work. Only relative changes in the entropy within individual clusters provide reliable signals for error detection.

Second, within a cluster, the in-cluster entropy can be very volatile. The empirical entropy is only a rough estimate computed from a sample; if the sample size is small (*e.g.*, when a cluster has only eight metrics), changes in in-cluster entropy may not indicate anomalies. For example, changes before time-interval 56 in Figure 8.15 are normal.

Third, judgment based on a single observation is not reliable(see, *e.g.*, time-interval 3, 10 and 35 in Figure 8.14). An algorithm must consider several samples before deciding that an error exists; otherwise, many false alarms may be raised.

These observations suggest that a deviation in entropy is a reliable indication of errors only if the deviation is relative and persistent.

Therefore, we choose to employ the Wilcoxon Rank-Sum test to identify significant shifts in the in-cluster entropy of individual clusters. The Wilcoxon Rank-Sum test suits our needs, as 1) it is non-parametric (*i.e.*, we make no distribution assumptions), and it does not learn a threshold or rely on one to work; 2) it is a statistical test, which allows temporary fluctuations to be accommodated.

7.4.2 Error Detection by Wilcoxon Rank-Sum Test

For error detection, let E_i be the in-cluster entropy of cluster E at time i. In order to detect a significant change in E_i when a fault occurs, we keep two sliding windows of E_i's. The test window consists of the most recent n E_i's. The baseline window consists of the m E_i's preceding the test window. We apply the Wilcoxon Rank-Sum test to the two windows. If the test indicates a significant shift between values in the two windows, an alarm is raised.

The Wilcoxon Rank-Sum test is a well-established hypothesis test. In our case, the null hypothesis is that the two sample sets $\{E_{s+1}, E_{s+2}, ...E_{s+m}\}$ and $\{E_{s+m+1}, E_{s+m+2}, ...E_{s+m+n}\}$ from the two sample windows $(s + 1, s + m)$ and $(s + m + 1, s + n)$ are from the same distribution. Then the test is done with the method introduced in Section 5.4.

We concurrently monitor each cluster at each time interval. At any time, if no cluster reports an anomaly, we consider the system to be in a healthy state. If any cluster reports an anomaly, we consider an error present and raise an alarm.

Chapter 8

Evaluation

In this chapter we present the evaluation of our two solutions. We use the same testbed in one of the previous work by Munawar [65]. Therefore, in order to make this thesis self-contained, we reproduce the evaluation approach in Section 8.1 before we present our evaluation.

8.1 Evaluation Approach

In this section we describe the setup we use and the methodology we follow to evaluate the feasibility and effectiveness of our solution approach. The setup essentially refers to one or more software systems that require monitoring and a managing system which monitors those systems. We use a systematic approach to study the algorithms and methods we devise for system modeling and monitoring.

There are two important premises that underlie or work. First, distributed, transaction-oriented software systems are complex. Second, monitoring these systems is costly both in terms of the monitoring overhead and the human involvement required. It is thus necessary to choose an evaluation setup that matches these premises. Two choices are available in this regard: production systems and experimental test-beds.

A system in production is one that is in actual use, providing real services. Obtaining access to production systems for research purposes is problematic for a variety of reasons. These systems manage sensitive information and provide critical functionality to organizations. Access to third parties raises concerns regarding sensitive and private data. System operators also frown upon any activity that risks affecting system reliability. Our solution approach requires collecting much more

data than what is collected by default in most software systems. System operators will be reluctant to subject their systems to the resulting adverse performance impact.

Several organizations have made web server access logs available to the research community (see, *e.g.*, [47]) However, this data only allows the workload and the user access patterns to be studied. Moreover, only post-mortem analysis can be performed on such data. Our work relies on management metrics, which are much richer than what the access logs contain.

To investigate the effectiveness of our solution approach, we not only need access to the monitoring data, but we also need to have the ability to control the data collection. We, therefore, choose to build our own experimental test-bed. This is described next.

Target Platform

The prevalence and complexity of multi-tier, component-based software systems make them an ideal target for our research. To this effect, we use a Java EE-based software system as our target system. This system is built using the WebSphere application server [28], which provides the execution engine for Java EE applications. To support long-term data persistence, we make use of the DB2 [29] database management system. Both WebSphere and DB2 are industrial-strength products that have significant shares of their respective markets. Our choice is motivated particularly by the fact that WebSphere provides advanced management interfaces; in particular, it allows dynamic, fine-grained control of metric collection. DB2 also provides advanced monitoring facilities, albeit at a coarser granularity. While conceptually simple, our target system displays significant internal complexity. Both the application server and the database server implement complex functionality and provide many advanced features.

A simple test-bed based on these products is shown in Figure 8.1. All entities are connected via a Gigabit LAN. The setup in Figure 8.1 can be scaled up by adding more application servers or databases, and separate web servers. In related work we have extended this basic setup to include multiple application servers.

Applications

We use our target platform to execute several existing applications that mimic functionality implemented in real transaction-oriented software systems. Although

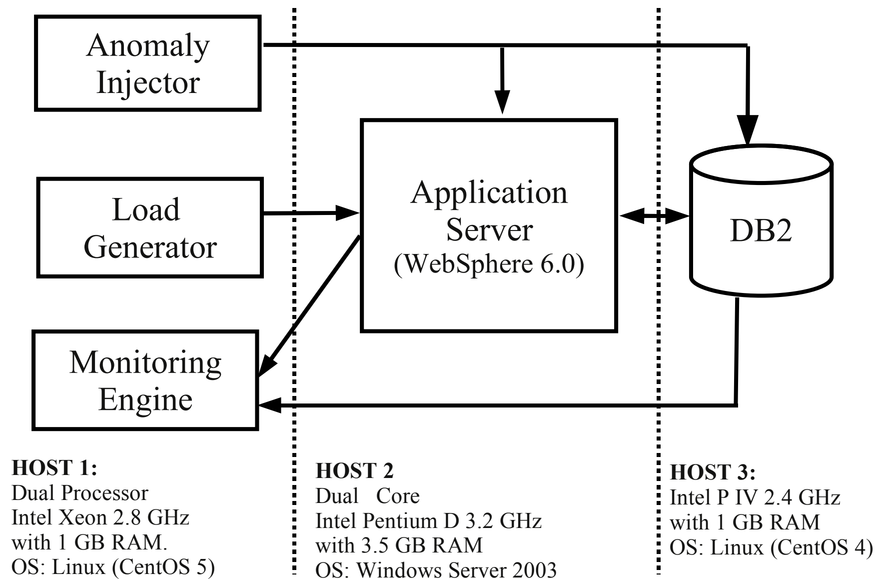


Figure 8.1: Experimental setup

these applications vary in size and functionality, and have been developed by different organizations, they share common characteristics. First, they have been built using the Java EE framework and provide a web-based user interface. Second, they require the use of a database management system. Most of these applications have been designed for the performance benchmarking of web transaction systems.

- PlantsByWebSphere PlantsByWebSphere [30] is a Java EE application developed by IBM to showcase the features and capabilities of the WebSphere application server. It implements an online store, selling plants and gardening tools. It allows users to create accounts, browse, check items of interest in detail, and purchase items. The application is built using standard Java EE components such as EJB, Servlet, JSP, and message-driven beans.
- RUBiS RUBiS [69], originally developed at Rice University, is a performance benchmarking application, which implements an online auction site similar to eBay. Its workload consists of web interactions for selling and browsing items, bidding, bids and ratings tracking, and handling user comments. In our setup we use a servlet-only implementation of RUBiS.
- TPC-W TPC-W [84] is a performance benchmark specification designed to evaluate web-based transaction-oriented systems. We use a servlet-based implementation of the specification. The application implements the functionality of an online retail store, allowing users to browse and purchase items.

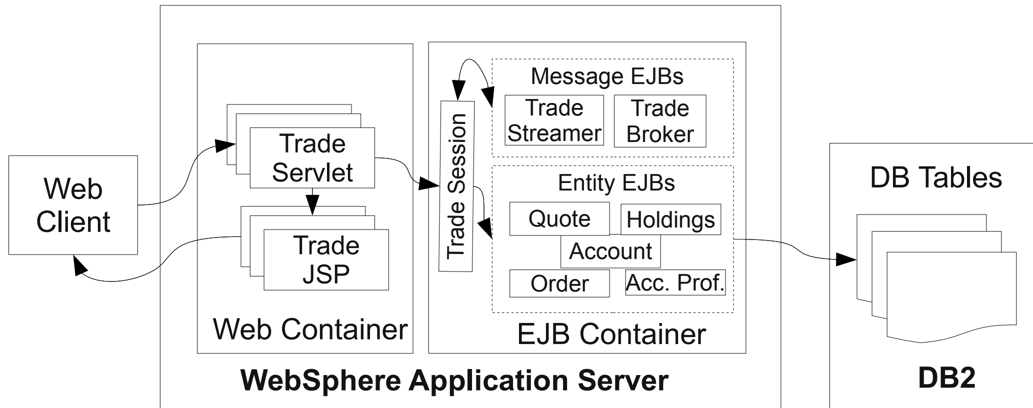


Figure 8.2: Overall structure of the Trade application

The benchmark can be configured to use workload profiles, which correspond to different proportions of “browse” to “buy” web interactions.

- Trade Trade [14] is a Java EE application developed by IBM that implements a stock brokerage system. The application allows end-users to trade securities. For example, users can register themselves, view stock prices, buy and sell stocks, check their accounts, and track their orders. It has been designed to exercise many features of the WebSphere application server. It is built with components such as EJB, Servlet, and JSP. It also makes use of the Java Database Connectivity (JDBC) to access the database management system and the Java Messaging Service (JMS) for asynchronous order processing. The main components of the Trade application are shown in Figure 8.2. A web interaction in Trade involves many components, even without taking into account the components of the underlying platform. While it is possible to access the Trade application via a native or a web service interface, we only use the web interface, which clients can access via a browser.

  We employ Trade as our main target application because of its large size and its use of the many features of the Java EE technology. Trade comprises many more components than the other applications; it is thus a better candidate to evaluate our monitoring and diagnosis solution approach. We use the other applications to validate our claim that stable metric correlations exist in software systems.

Workload

Many aspects of our work depend on observing a system in operation. We create synthetic workloads by simulating a population of users accessing the functionality provided by the applications. We use an open-loop workload[1] to estimate the effect of monitoring on system performance. We make use of a closed-loop workload for all other experiments.

For Trade and PlantsByWebSphere, we use our own workload generators, which gives us the flexibility to generate different load patterns. By default, we use a random uniform load pattern, which is configured to cause the system to operate over a wide range of resource utilization levels. For RUBiS and TPC-W, we use the emulated clients that are provided with these benchmarks. Our workload generators execute on a separate machine, and we ensure that enough resources are dedicated to avoid bottlenecks in the client machine.

Monitoring Engine

Our monitoring engine consists of data-collection and data-analysis engines. It also contains a model repository. The monitoring engine operates from outside the target system and executes on a separate host.

The data collection engine manages the collection of metric data from the target system. This data is either processed online or saved in a local database for offline analysis.

The metric data originates from the subsystems of the target system. We use the JMX interface to collect metrics from the WebSphere application server. We use the DB2 Snapshot interface to collect metrics from the database. The workload generators also expose metrics, which we collect through log files. For collecting host-level metrics, we use the WMI interface on windows hosts or the `sar` utility on Linux hosts.

We collect metric data at a fixed rate, which we set to 10 seconds. This choice allows the overhead of collecting a given set of metrics to remain low, while having sufficient resolution to capture dynamics of interest in the target system. The transactions in our applications are short-lived; when the system is not overloaded,

[1] In an open system the arrival of new requests is independent from the completion of other requests. In a closed system new requests are submitted upon completion of previous requests, and the load is primarily a function of the clients.

Component	Metrics
Web Container	# Sessions created/invalidated
Thread Pools	#Threads created/active, free pool size
JDBC module	Response time, #Free connections
Servlet/JSP and EJB	#Requests, #Instantiations, Response time
Database	#Active connections, #Log writes
Database tables	#Rows retrieved/written/deleted

Table 8.1: Examples of metrics collected

most transactions take much less than one second to complete. As such, a 10-second interval allows significant activity to be captured. Furthermore, our choice of collection interval allows prompt detection of anomalies in the monitored system.

The data analysis engine is responsible for processing the collected data. The processing involves either learning models from the collected data or checking new data using the learned models. Our analysis engine is built in Java. We leverage the implementation of regression models available in the Weka-3 data mining [86] package. However, the majority of the analysis engine is custom-built. This includes tests for checking model assumptions, the correlation identification and validation logic, the metric selection methods, the diagnosis method, *etc.*

Monitoring Data

Our data comprises periodically-collected management metrics from WebSphere and DB2. For example, with the Trade application, the raw data sets consist of more than 600 metrics collected every 10 seconds. We take some basic filtering steps to discard metrics that provide little information or are redundant. More specifically, we check whether the metrics display non-zero variance in a small window of samples; we use a window of 60 samples in our experiments. Though not necessary in general, we discard metrics that we find to be redundant based on naming conventions. For example, if two metrics are collected, we would ignore a metric that represents their sum. From the metrics we collect from Trade, only 352 metrics remain after the basic filtering. Table 8.1 lists a few examples of metrics included in our data sets.

Experiment Framework

We have developed a scripting framework to coordinate our experiments. It consists of an experiment controller and daemons running on hosts involved in the experiments. The controller script sends commands for the daemons to execute. These commands include operations to reset state, to inject faults, to start and stop the database and the application servers, to enable and disable metric collection, and to start and stop workload generation.

All our experiments involve preparatory steps such as synchronizing time, restarting the application and database servers, resetting application and database states, and warming up the target system.

8.1.1 Methodology

Are metric correlations stable? Can we detect fault-induced disturbance with correlations? How well can we localize faulty components with correlation? To answer these and other questions raised by our solution approach, we design and carry out controlled experiments using our test-bed. More specifically, we carry out two types of experiments: *normal activity experiments* and *fault-injection experiments*. Normal activity experiments involve studying the system under normal operating conditions. These experiments are used to characterize the target system's normal behavior, to check the robustness of our modeling approach, and to assess the overhead of monitoring. These experiments are typically long (spanning several hours) to make analysis less vulnerable to spurious observations. Fault-injection experiments are relatively much shorter (lasting less than an hour) and are discussed next.

8.1.2 Fault Injection

The purpose of our fault-injection experiments is to study how well we can detect faults and how accurately we can localize the faulty components. To this end, we postulate various types of faults that can occur in a system. We inject the faults into the target system while it is in a healthy state and examine the response of the monitoring system. Knowing the ground truth about the faults, we check whether the monitoring system can detect the faults. Likewise, knowing the components in which the faults exist, we can measure the accuracy of the diagnosis produced.

Fault Class	**F**ault Category	**N**umber of **C**omponents **I**njected
Application faults	Exceptions in JSP and EJB components	12
	Delays in JSP and EJB components	12
	Locking in DB tables	5
Operator mistakes	Misconfigurations	3
	Deletion of JSP components	7

Table 8.2: Summary of the faults injected

Faults can be defined at different granularity. We can create faults that cause subsystems to fail (*e.g.*, kill a database or application server process, disconnect the network, *etc.*). These faults cause major subsystems to stop completely and thus can be detected easily by probing the specific subsystems. However, with such coarse-grained faults we cannot assess the effectiveness of our diagnosis approach at the level of software components.

We have implemented faults at the level of software components (*e.g.*, application components, middleware components, database tables, *etc.*). Most of these faults cause the target system to fail partially, making them more difficult to detect and diagnose. With such faults, we can evaluate the effectiveness of our solution approach in the presence of a system's internal complexity and dynamism.

The fault injections we have designed can be broadly grouped into two categories: application faults and operator faults. We simulate operator faults, because it has been observed that a large proportion of faults in software systems are operator faults [67, 83]. In each category, we have several classes of faults. A summary of the faults we use in our experiments is given in Table 8.2 and further details are provided in the following sections.

Application Faults

These faults are injected in application components, which causes the execution of the application to be affected directly. Such faults may arise from faulty implementation, which may have escaped testing or may have been introduced during a system update. Such faults may also be caused by faulty logic, which may cause part of the application to under-perform or even stall.

Faulty execution flow: This class represents faults that cause components to deviate from the normal flow of execution. We instrument the target application to induce two types of faults: *unhandled exceptions* and *null call returns*. Exception faults involve throwing an unhandled exception with probability e_{prob} when a selected method of a component is executed. Null returns are similar to the exception faults except that they cause a selected method of a component to return null instead of throwing an exception.

The effects of both types of faults are similar in our test-bed, as most cases of null returns cause exceptions. We thus only discuss results of the exception faults in our evaluation.

Performance degradation: This class of faults causes slow-down in specific application components. We modify the target application to introduce two types of such faults: *delay loops* and *thread sleep*. Delay-loop faults entail delaying completion of a selected method for d_{len} time units by executing extra cpu-intensive logic. To configure these faults, we specify a component, one of its methods, the delay-loop duration d_{len}, and a probability of activation, d_{prob}, when the selected method is executed. Thread sleep is similar to delay loops except for the fact that thread sleep causes the executing thread to sleep for d_{len} instead of keeping the processor busy.

Both types of faults cause delays in application components. However, unlike thread sleeps, delay loops tend to monopolize the CPU on the application host, causing widespread disturbance in the system. Much more insight can be had by analyzing effects of thread sleeps; we thus limit our evaluation to such faults.

Database table locking: This class of faults represent external disturbance to components in the database used by our application. We simulate table-locking faults which periodically lock a chosen database table. The lock is activated for l_{lock} fraction of every l_{interval} time interval during the fault-injection period.

In our experiments we configure our application faults using the parameters listed in Table 8.3. The tasks of error detection and diagnosis are more difficult when faults are probabilistic rather than deterministic. Probabilistic faults are not unrealistic; for example, in a load-balanced, clustered system a fault that affects a member of a cluster is likely to have effects similar to that of a probabilistic fault.

Parameter	*Value*
e_{prob}	0.3
d_{prob}	0.2
d_{len}	2000 (ms)
l_{interval}	1000 (ms)
l_{lock}	0.5

Table 8.3: Fault parameters

Operator Faults

These faults simulate mistakes by a system operator during configuration or tuning of the system. The faults we devised include misconfiguration of credentials in the application server for database authentication, wrong tuning of system components such as connection and thread pools (*i.e.*, the pool sizes are set too low), and deployment faults such as inadvertent deletion of application components.

The specifics of this class of faults are as follows:

- **JSP deletion:** the fault consists of removing JSP files from the deployment files. We consider the separate removal of seven different JSPs. These faults cause user requests to fail when a missing JSP is involved.
- **Thread pool size too low:** the fault entails setting the maximum size of the main thread pool of the application server to a low value. This limits the application server's ability to accept and perform concurrent work.
- **Database connection pool too small:** the fault entails setting connection pool size in the application server to a low value. The fault causes a slow down in retrieving data from the DBMS.
- **Database authentication error:** the fault involves using wrong credentials for the application server to authenticate with the database. This fault completely prevents the application server from fetching persistent data from the database.

8.1.3 Fault-Injection Experiments

Each of our fault-injection experiments lasts for approximately 30 minutes, with a fault injected at the 20-minute point. We treat the first 10 minutes as a warm-up

period, and ignore it. Thus, from the perspective of our analysis we have a fault injected at the 10-minute mark, or at time-interval 56. Any error reported prior to time-interval 56 is a false alarm. In order to keep the false positive rate low, we use a window size of 12 for all the Wilcoxon Rank-Sum test, which is the largest value typical in pre-computed statistics table for small samples. Therefore, any alarm reported within 12 sampling periods from the point of fault injection is considered a successful detection.

8.2 Evaluation of Linear Modeling

We first present the evaluation for our linear modeling techniques. The system modeling process yields 988 OLS models and 3219 GLS models. We first evaluate the performance of these models and then present the error detection results with these models.

8.2.1 The Performance of Individual Models

We first evaluate the performance of individual models we have learned during the system modeling step.

Based on our assumption that the metric correlation are stable when there is no fault, a model is expected to report no outliers before the fault is injected for each of the 39 error-injected experiments. We also assume that some of the correlation would break when there is a fault present in the system, therefore we expect some models to report many outliers after the fault is injected.

According to the expectation, the performance of models could be categorized in three types as shown in Table 8.4. Non-informative models do not report many outliers throughout the experiment. Informative models may have a few false positives when there is no fault; but have many outliers reported when the fault is present. Some models report many outliers even when there is no fault, which we consider to be inaccurate models. For the seek of evaluating the performance of models, we consider a false positive rate up to 5% is possible for a good linear model. Therefore, we use 5% as the threshold to determine if there is "many" outliers reported in Table 8.4.

Considering the performance of each model in the 39 experiments, we divided models into a few categories according to the definition in Table 8.5.

	Outliers reported before fault occurs	Outliers reported after fault occurs
Non-informative	Less than 5%	Less than 5%
Informative	Less than 5%	More than 5%
Inaccurate	More than 5%	Any

Table 8.4: Model performance definition

	Non-informative	Informative	Inaccurate	Number	Proportion
Non-informative Models	39	0	0	298	7.1%
Informative Models	Any	1+	0	3671	87.3%
Inaccurate Models	Any	0	1+	179	4.2%
Not any of the above	Any	1+	1+	59	1.4%

Table 8.5: Model performance

There are 298 or 7.1% non-informative models, which never report many outliers in the 39 experiments. Those models probably represent inherent correlation in the system which never break.

There are 3671 or 87.3% informative models, which are never inaccurate in any of the 39 experiments and are informative in detecting errors in at least 1 of the 39 experiments. This conforms our assumption that many of the linear correlations in the system are stable, but may break when fault is present.

There are 179 or 4.2% inaccurate models, which report many outliers when there is no fault during at least 1 of the 39 experiments and are never informative in any experiments. For some reason those models are no longer valid after we learned them in the system modeling step.

There are 59 or 1.4% models show unexpected behaviors. They are sometimes inaccurate and sometimes informative. Such behaviors are not expected. However, given the small number of them, we believe they are probably just caused by some random errors in the modeling and statistics.

In sum, 95% of the models are stable correlations and most of them are useful in detecting some errors.

8.2.2 Error-Detection Examples

We first show a few examples where we successfully detect errors with our models. These example support our claim that many errors in the system can invalidate many linear correlations between metrics such that the increased number of broken models can be used to identify errors in the system. In addition, these examples

also show the complexity of automating the error-detection process and illustrate the benefit of applying the Wilcoxon Rank-Sum test.

Figure 8.3, 8.4, 8.5, 8.6 and 8.7 show the number of models that report outliers during five experiments separately. Due to space limitation, we will not show such figures for all 39 faulty experiments but these five experiments are very representative.

In all five experiments, before the fault is injected at time 56, the number of broken models varies around 100 or 2.5% of the 4207 models. Those outliers may be caused by two reasons. First, there may be some difference in the environment between the learning experiment and the faulty-injection experiment. As a result, a few models may no longer be valid. Second, a small false positive rate of one or two percent is fair since by definition of confidence interval the dependent variable may have a small chance to be out of the predicted confidence interval.

In four of the five experiments, we can observe an increase in number of models broken when the fault is injected at time 56. We successfully detect these four errors with the Wilcoxon Rank-Sum test. The only exception is the one presented in Figure 8.5, which is not detected.

We can learn a few points from these examples.

First, it is possible that some of the learned models will not hold in the new environment, where the models are being used for monitoring. Therefore, the method to detect errors must be able to tolerate such invalid models. This can be done by the Wilcoxon Rank-Sum test, which only detects changes or increases in number of broken models instead of the absolute number. Therefore, no anomaly is reported from time 0 to time 56 in all five experiments. At time 56, the fault is injected. The number of broken models show an increase in four of the examples, which can be easily identified with the Wilcoxon Rank-Sum test. Without such a test, we would need to set *a priori* an appropriate threshold for the number of broken models (*e.g.*200 in our example), which is non-trivial.

Second, we have found that it is common for some models to break occasionally. As a result, the number of models that report outliers may suddenly increase at some point and then drop shortly afterward. A threshold-based detection scheme will likely cause false alarms at these points. For example, at 48 in Figure 8.6, models reporting outliers show a sudden and temporal increase for two consecutive samples. It is challenging to avoid false positives at these points. With the Wilcoxon Rank-Sum test, however, we can make detection robust to the temporary spikes, thus avoiding false positives.

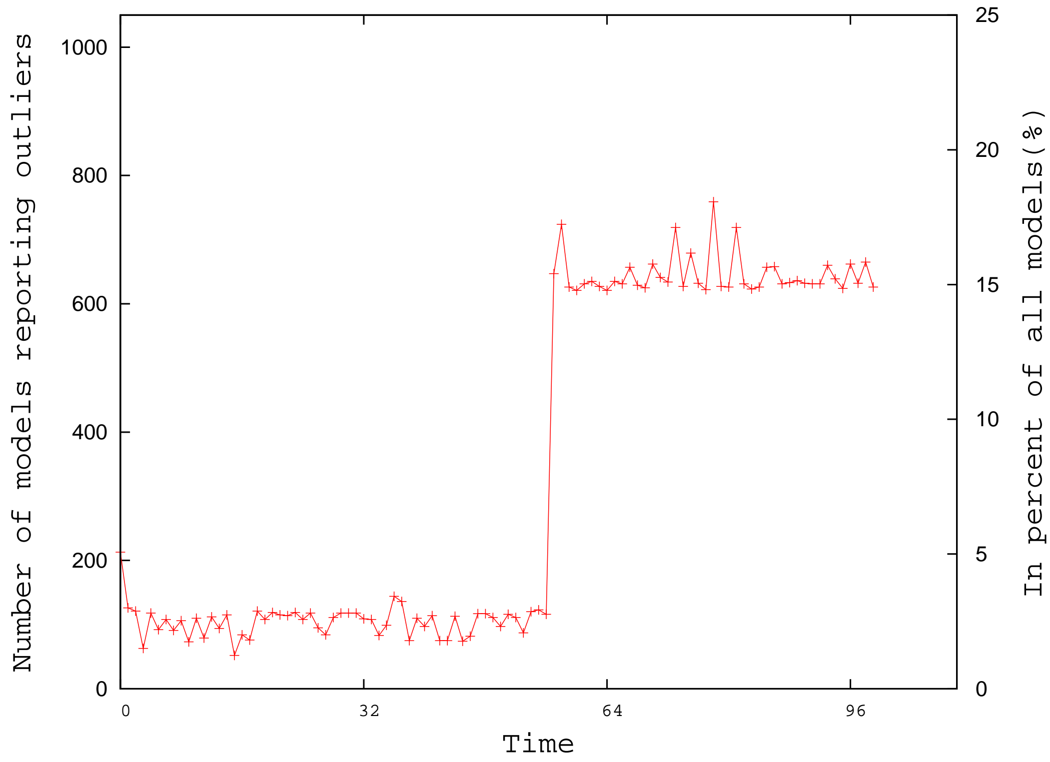


Figure 8.3: Sample fault detection - Mis-ds-authentication

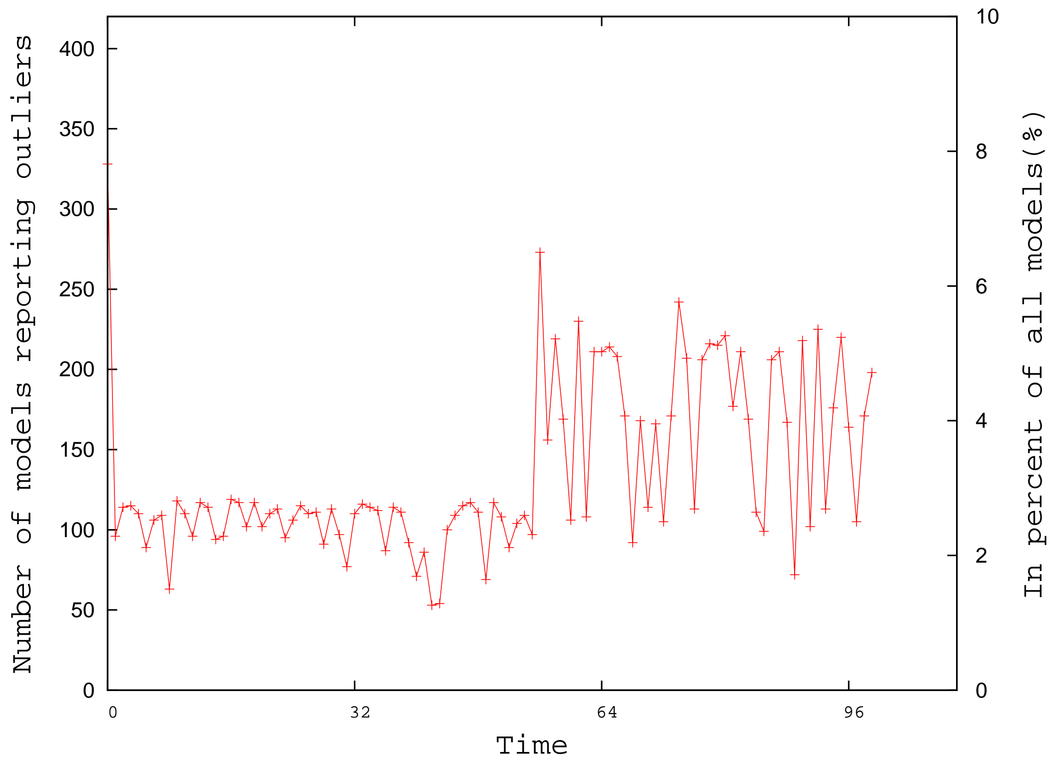


Figure 8.4: Sample fault detection - Mis-ds-connection-pool

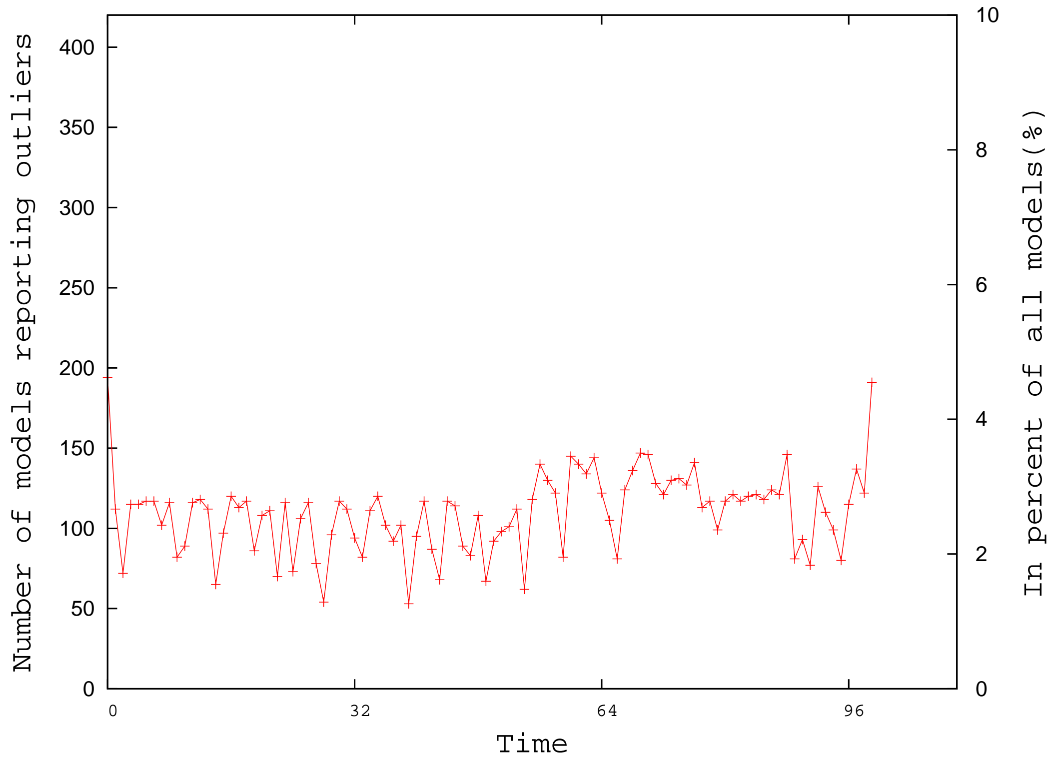


Figure 8.5: Sample fault detection - Del-AccountJSP

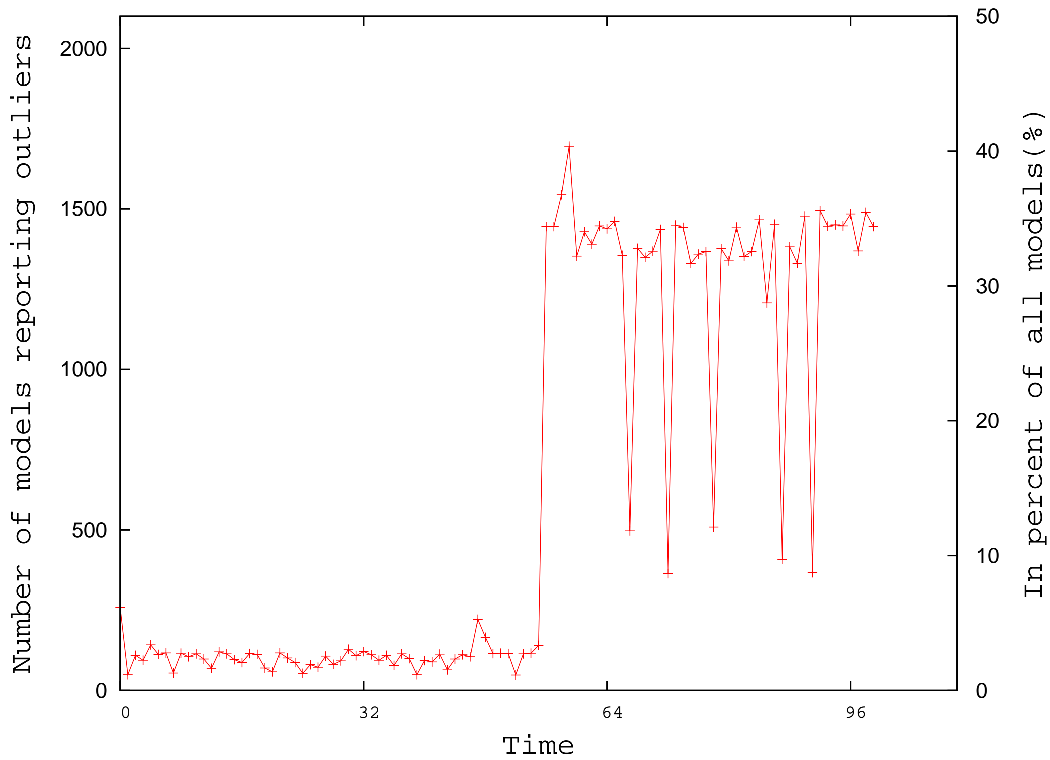


Figure 8.6: Sample fault detection - Del-DisplayQuoteJSP

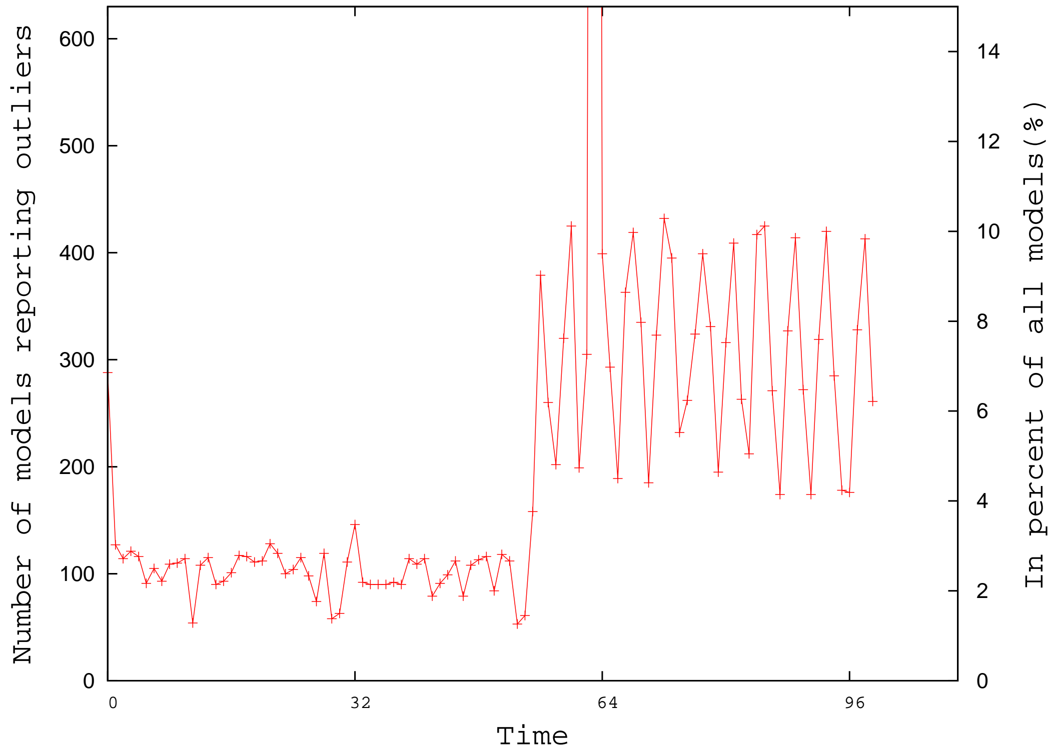


Figure 8.7: Sample fault detection - DB-QuoteEJB

8.2.3 Experimental Results

We list the first time our monitoring system report an outlier for the 39 experiments as well as the rank of the faulty component in the diagnosis in Table 8.6. We place "X" when error is never reported and "-" when the faulty component has an anomaly score of 0.

To evaluate the performance of the error detection algorithm, we use the measurements outlined in Section 4.1.1. The details of the measurements for our evaluation based on the general idea in Section 4.1.1 follows.

An ideal fault-detection algorithm should report anomalies whenever there is a fault and report nothing otherwise. Nevertheless, after a fault occurs, the current observation window may contain both normal and anomalous samples. Thus, the Wilcoxon Rank-Sum test on sliding windows may not report anomalies immediately.

As a result, there may be a time lag up to the length of the sliding window before anomalies are detected. We call this lag a *detection window*, which equals the size of the observation window times the duration of the sampling period. As such, we set the following requirements for our algorithm:

1. *Recall:* when a fault occurs, the algorithm should report an anomaly after the fault occurs and before the detection window elapses.
2. *Precision:* any error reported should truly represent a fault. Any error reported when there is no fault is considered a false alarm and should be avoided.

Therefor, for each fault-injection experiment, the error is considered detected (counted as a *true positive*) if there is no false alarms before the error is present, and there is at least one alarm soon after the error is present. If there is any alarm before the error is present, it is counted as a *false positive*. If there is no alarm reported but there is an error present, it is counted as a *false negative*. If there is no alarm reported and there is no error present throughout the experiment, it is counted as a *true negative*.

We then calculate the recall, precision, false positive error rate and F-measure according to the definitions in Section 4.1.1. These measurements are reported in Table 8.8 under the column "Current". We successfully detect the errors in the system in 33 out of 39 experiments with different faults injected. In addition, our algorithm does not report any false alarms before the fault is injected in any of the 39 experiments. All false positives made by individual models have been successfully filtered at the error detection step. Given a zero false positive error rate, it is impressive that we still have a high recall of 85%. Therefore, we have a high F-measure of 0.92.

The fault diagnosis is also informative. Although we are not able to pinpoint the faulty component in most experiments, we still manage to rank the faulty component in top 14 out of 28 components in 20 of the 33 experiments when we detect errors. This diagnosis illustrate some minimal information metric correlation can provide regarding the fault localization. We are expected to explore methods to improve the diagnosis in the further work.

8.2.4 Comparison with Prior Work

We would like to compare our new methods with prior work [63, 64, 62]. To this effect, we use the same data set to compare the methods. We first apply ordinary least squares to learn linear models, and select linear models with $R^2 > 0.95$, without any heteroscedasticity test. This procedure yields 15228 models. We then apply these models to the data from fault-injection experiments and check whether they are useful in detecting faults. Methods in prior work rely on a threshold of the number (or proportion) of models that need to break before errors are detected.

	Error detection time	Faulty component rank
Mis-ds-authentication	63	16
Mis-ds-connection-pool	68	14
Del-AccountJSP	X	
Del-DisplayQuoteJSP	57	-
Del-TradeHomeJSP	65	12
Del-MarketSummaryJSP	66	-
Del-OrderJSP	68	-
Del-PortfolioJSP	65	1
Del-QuoteJSP	66	2
Mis-ThreadPool	X	
DB-QuoteEJB	66	2
DB-HoldingEJB	66	1
DB-OrderEJB	65	3
DB-AccountEJB	64	-
DB-AccountProfileEJB	65	2
Exception-QuoteEJB	66	2
Exception-OrderEJB	65	5
Exception-HoldingEJB	64	9
Exception-AccountProfileEJB	66	3
Exception-AccountEJB	64	5
Exception-MarketSummaryJSP	65	-
Exception-QuoteJSP	66	1
Exception-PortfolioJSP	X	
Exception-WelcomeJSP	68	-
Exception-AccountJSP	70	-
Exception-OrderJSP	X	
Exception-TradeHomeJSP	62	13
Thread-QuoteEJB	63	2
Thread-OrderEJB	65	4
Thread-HoldingEJB	62	10
Thread-AccountProfileEJB	65	9
Thread-AccountEJB	66	10
Thread-MarketSummaryJSP	66	-
Thread-QuoteJSP	65	-
Thread-PortfolioJSP	65	-
Thread-WelcomeJSP	62	-
Thread-AccountJSP	65	-
Thread-OrderJSP	X	
Thread-TradeHomeJSP	66	-

Table 8.6: Error detection and fault localization with linear models

We experiment with different threshold values and report the results under column with "1%" to "20%" in Table 8.7. A summation is reported under the same column in Table 8.8.

The results show that fact that there are many false alarms if the threshold is too low and many faults are not detected if the threshold is set higher. This suggests that setting static thresholds is problematic. Not only it is difficult to select an appropriate threshold beforehand, it is even doubtful that an appropriate static threshold exists. For example, comparing the results in Table 8.8, we conclude that if we are most concerned with the false positive error rate, the best static threshold is 5%, when 16 out of 39 faults are detected without any false alarms. If we are most concerned with the F-measure, the best static threshold is 5%, when 25 out of 39 faults are detected, and in 3 out of 39 fault-injection experiments we have false alarms. Our current method works much better by detecting 33 out of 39 faults without any false alarms.

To address the problem of selecting an appropriate threshold and make it possible to compare our models with models in prior work (*i.e.*, we would like to compare our OLS and GLS models to the previous OLS model with no heteroscedasticity test applied), we apply the Wilcoxon Rank-Sum test in both cases and see how they compare in terms of error detection. We report the results under column "WRS test" and "Current" in Comparing the last column in Table 8.7 and Table 8.8., we see that models from prior work has 5 false positives. This is understandable; as shown in Figure 6.2 and Figure 6.6, the prediction intervals do not account for heteroscedasticity, and thus could be too narrow, falsely labeling normal samples as outliers. In addition, with models from prior work only 30 faults are detected successfully, which is less than the 33 faults detected with current work.

8.3 Evaluation of Information-Theoretic Solution

In this section we present the evaluation for our information theoretic modeling techniques. We first evaluate our similarity measure Normalized Mutual Information, and then evaluate the information theoretic solution.

8.3.1 Identifying Non-linear Correlations

We first study if the Normalized Mutual Information (NMI) similarity measure helps identify both linear and non-linear relationships between metrics. To bet-

Threshold to raise alarms	1 %	2 %	3 %	5 %	10 %	20 %	WRS test	Current
Mis-ds-authentication	0	56	56	57	57	X	64	63
Mis-ds-connection-pool	56	57	X	X	X	X	64	68
Del-AccountJSP	44	44	44	99	99	X	68	X
Del-DisplayQuoteJSP	47	56	56	82	93	X	66	57
Del-TradeHomeJSP	56	56	56	56	56	56	64	65
Del-MarketSummaryJSP	56	56	56	56	56	56	66	66
Del-OrderJSP	61	X	X	X	X	X	65	68
Del-PortfolioJSP	32	74	X	X	X	X	66	65
Del-QuoteJSP	25	25	56	56	56	56	66	66
Mis-ThreadPool	56	X	X	X	X	X	X	X
DB-QuoteEJB	56	56	X	X	X	X	62	66
DB-HoldingEJB	56	62	X	X	X	X	66	66
DB-OrderEJB	56	92	92	92	93	X	64	65
DB-AccountEJB	56	87	X	X	X	X	64	64
DB-AccountProfileEJB	28	77	X	X	X	X	66	65
Exception-QuoteEJB	36	56	56	56	59	71	62	66
Exception-OrderEJB	38	56	56	56	56	56	65	65
Exception-HoldingEJB	55	56	56	56	56	56	61	64
Exception-AccountProfileEJB	56	56	57	58	62	X	65	66
Exception-AccountEJB	20	56	56	56	56	58	47	64
Exception-MarketSummaryJSP	56	56	56	56	56	56	63	65
Exception-QuoteJSP	56	56	57	58	85	X	62	66
Exception-PortfolioJSP	62	X	X	X	X	X	X	X
Exception-WelcomeJSP	43	64	X	X	X	X	38	68
Exception-AccountJSP	X	X	X	X	X	X	68	70
Exception-OrderJSP	97	X	X	X	X	X	X	X
Exception-TradeHomeJSP	52	56	56	56	56	56	63	62
Thread-QuoteEJB	56	56	57	X	X	X	66	63
Thread-OrderEJB	58	58	74	74	84	X	64	65
Thread-HoldingEJB	54	56	X	X	X	X	44	62
Thread-AccountProfileEJB	37	37	X	X	X	X	63	65
Thread-AccountEJB	X	X	X	X	X	X	65	66
Thread-MarketSummaryJSP	78	X	X	X	X	X	52	66
Thread-QuoteJSP	68	X	X	X	X	X	X	65
Thread-PortfolioJSP	X	X	X	X	X	X	66	65
Thread-WelcomeJSP	92	92	X	X	X	X	66	62
Thread-AccountJSP	29	X	X	X	X	X	64	65
Thread-OrderJSP	80	X	X	X	X	X	33	X
Thread-TradeHomeJSP	78	78	X	X	X	X	67	66

Table 8.7: Error-detection comparison

Threshold to raise alarms	1.00%	2.00%	3.00%	5.00%	10.00%	20.00%	WRS test	Current
True Positive	22	25	16	16	15	9	30	33
False Positive	14	3	1	0	0	0	5	0
False Negative	3	11	22	23	24	30	4	6
Recall	56%	64%	41%	41%	38%	23%	77%	85%
False Positive Error Rate	39%	11%	3%	0	0	0	14%	0
F-measure	0.58	0.74	0.57	0.58	0.55	0.37	0.81	0.92

Table 8.8: Error-detection summary

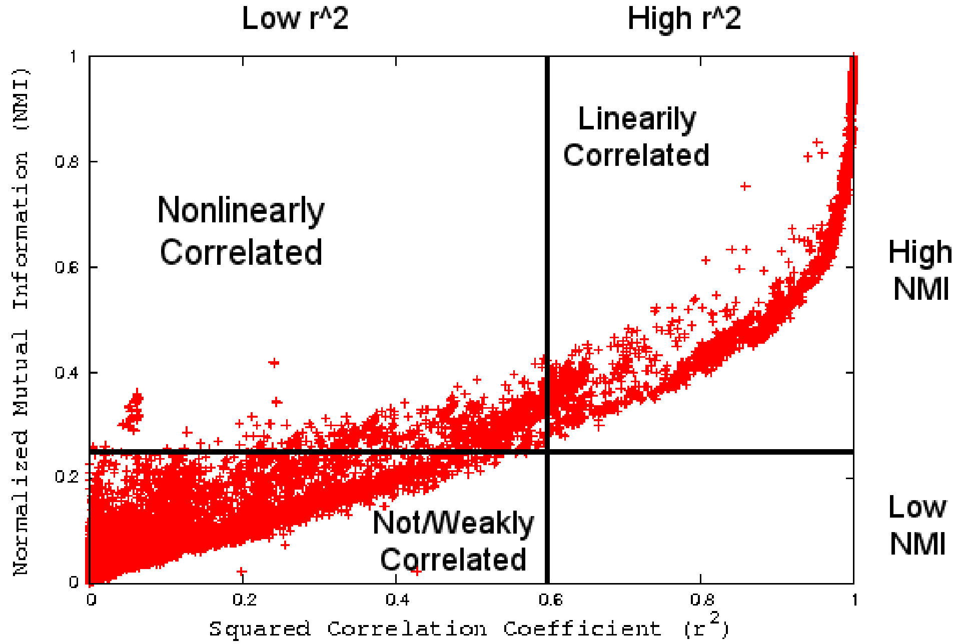


Figure 8.8: Metric-similarity measures

ter understand NMI, we compare it with the square of the Pearson's correlation coefficient, r^2, a measure of how strongly two variables are linearly correlated.

If metrics X and Y are linearly correlated, they should have high r^2 and high NMI; if they are correlated but the relation is nonlinear, they should have low r^2 and high NMI; if they are not correlated, they should have low r^2 and low NMI. Metrics will never have a high r^2 and a low NMI, as variables that are highly linearly correlated must be highly correlated, thus have a high NMI.

r^2 is a well-studied similarity measure. In general, $r^2 > 0.6$ indicates a strong linear relationship, and $r^2 > 0.9$ indicates a very strong linear relationship. However, NMI is a new similarity measure, proposed very recently, and has little guidance for what value of NMI constitutes an indication of strong relationship. We computed both r^2 and NMI for metrics pairs in a three-hour, fault-free experiment. The results, shown in Figure 8.8, indicate that for $r^2 = 0.6$, the NMI is at least 0.25. As such, we consider NMI > 0.25 to indicate a strong correlation. Similarly, we consider NMI > 0.5, which corresponds to $r^2 > 0.9$ for a linear correlation, to indicate a very strong correlation.

Filtering metrics pairs with $r^2 < 0.6$ (not strongly, linearly correlated) and NMI > 0.25 (strongly correlated) identifies metrics pairs which have strong nonlinear correlation. Our experiment shows that there are quite a few nonlinear

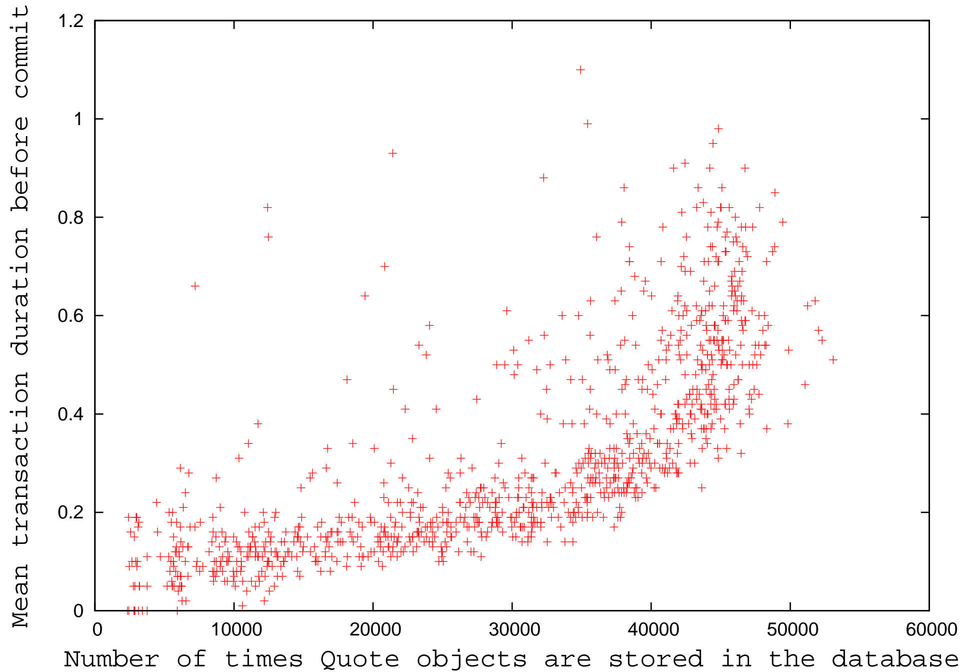


Figure 8.9: Metric relationship - a power function

correlations within the system. Two examples of such correlations are shown in Figure 8.9 and 8.10. Figure 8.9 shows the correlation between a response time and an activity metric. Beyond a certain level of activity (in this case, saving `Quote` objects to the database), the increase in the response time of related transactions becomes non-linear. Figure 8.10 show the correlation the same activity metric and the time taken to update and publish `Quote` objects. After a certain level of activity, the response time is roughly constant but noisy because of additional factors that affect response time. A conclusion can be easily drawn from the figure that if the number of times Quote objects are stored in the database exceeds 20000, the response time of the relevant method would unlikely be lower than 40 seconds. This relationship can hardly be identified with regular modeling unless a piecewise function is specifically designed for it. In our case, however, it is easily identified with a NMI greater than 0.25.

8.3.2 Clustering of Metrics

Clustering using NMI subdivides the metrics in a number of groups, each containing metrics correlated with each other. We observe that there are usually a small number of big clusters and many small clusters. The exact number of big clusters depend on the NMI threshold used to do the clustering. Usually, there will be three

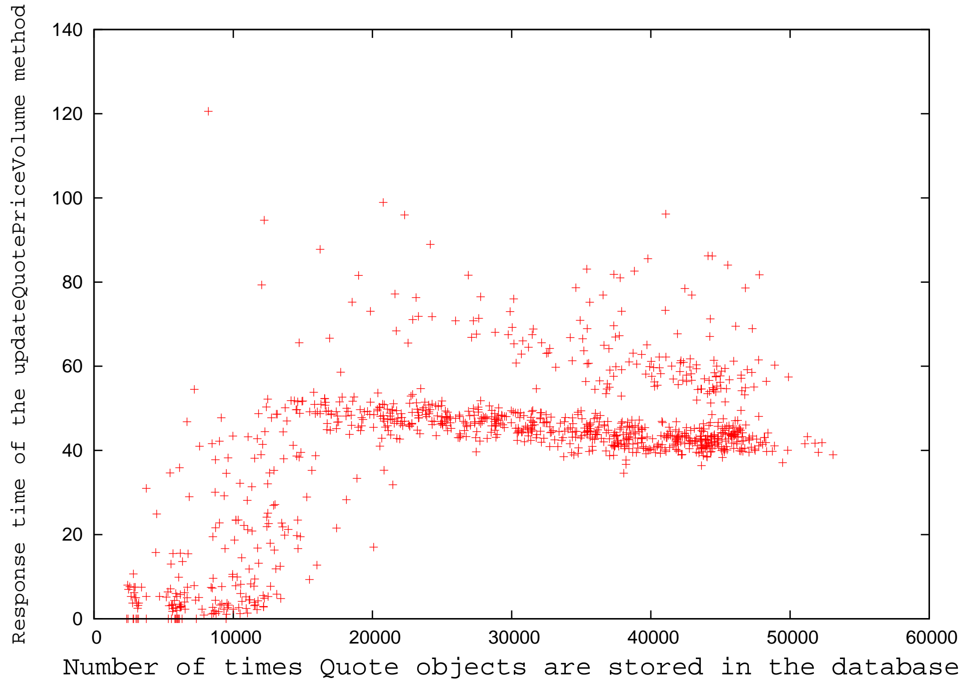


Figure 8.10: Metric relationship - a piecewise function

big clusters.

Given some domain knowledge, some of the big clusters obtained can be explained. For example, the largest cluster usually corresponds to the direct workload-induced effects on the system metrics. Most metrics in this cluster relate to activity count in the different components used by the Trade application. The second largest cluster also relates to workload-induced activity; however, it mostly contains metrics related to order handling and the messaging engine. Sometimes the two biggest cluster will merge as they are both workload related. The third largest cluster contains response-time metrics, whose behavior differ from that of activity metrics.

An alarm from a cluster with nearly a hundred metrics may be more valuable than an alarm from a cluster with only two metrics. The former involves a lot more metrics and it represents broader knowledge of the current system status. Moreover, since we have eight bins when calculate the in-cluster entropy, a cluster with too few metrics may not provide enough samples to estimate the empirical in-cluster entropy. Therefore, we typically require a cluster with at least eight metrics to be considered valid and only monitor these big clusters.

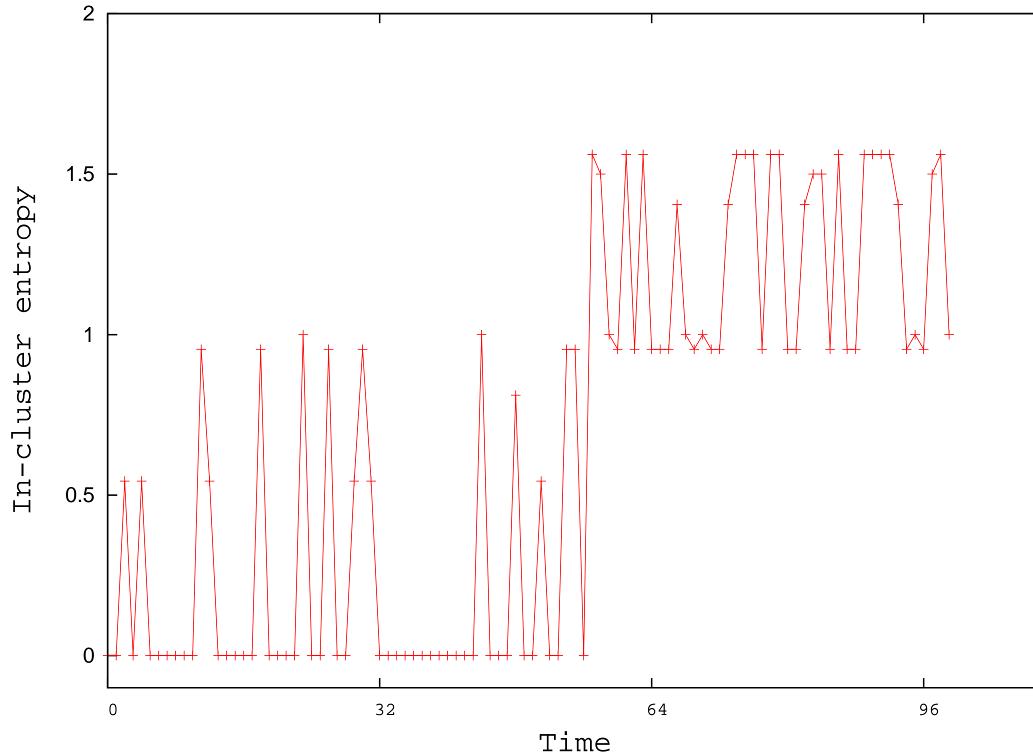


Figure 8.11: Sample in-cluster entropy

8.3.3 Error-Detection Examples

Similar to Section 8.2.2 we first show a few examples where we successfully detect errors with our models. These example support our claim that errors in the system may cause persistent change in the in-cluster entropy, which could signal an error in the system. They also illustrate why we use Wilcoxon Rank-Sum test to automate the error-detection process.

Figure 8.11, 8.12, 8.13, 8.14 and 8.15 show examples where the error in the system cause the the in-cluster entropy in some cluster to change.

In all five experiments, after the fault is injected at time 56, the in-cluster entropy show a persistent change which is detected by the Wilcoxon Rank-Sum test.

8.3.4 Experimental Results

We evaluate the detection performance using the 39 fault-injection experiments. We use the same definition for fault coverage and false positive as in Section 8.2.3. The first time an error is reported is presented in Table 8.9.

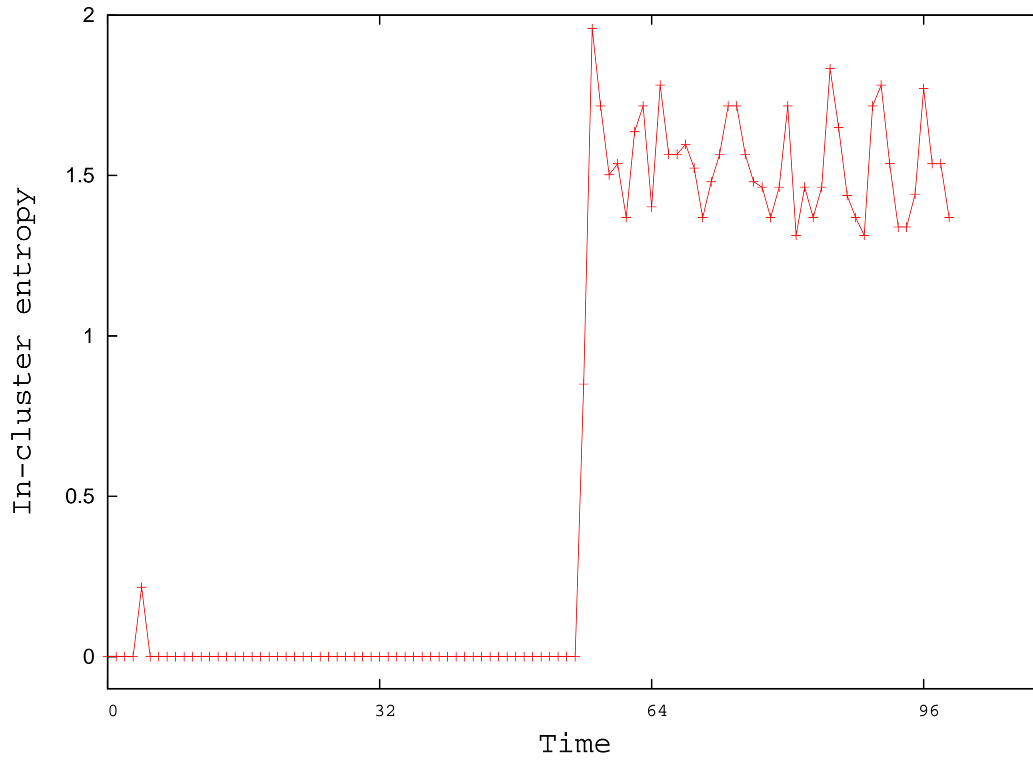


Figure 8.12: Sample in-cluster entropy

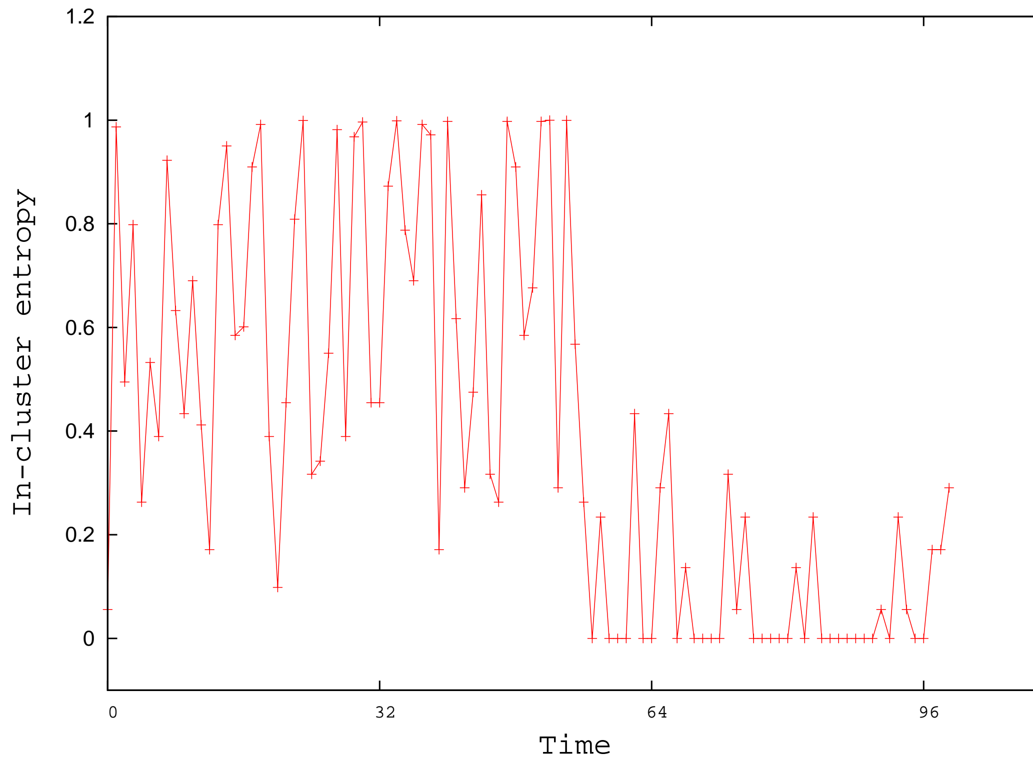


Figure 8.13: Sample in-cluster entropy

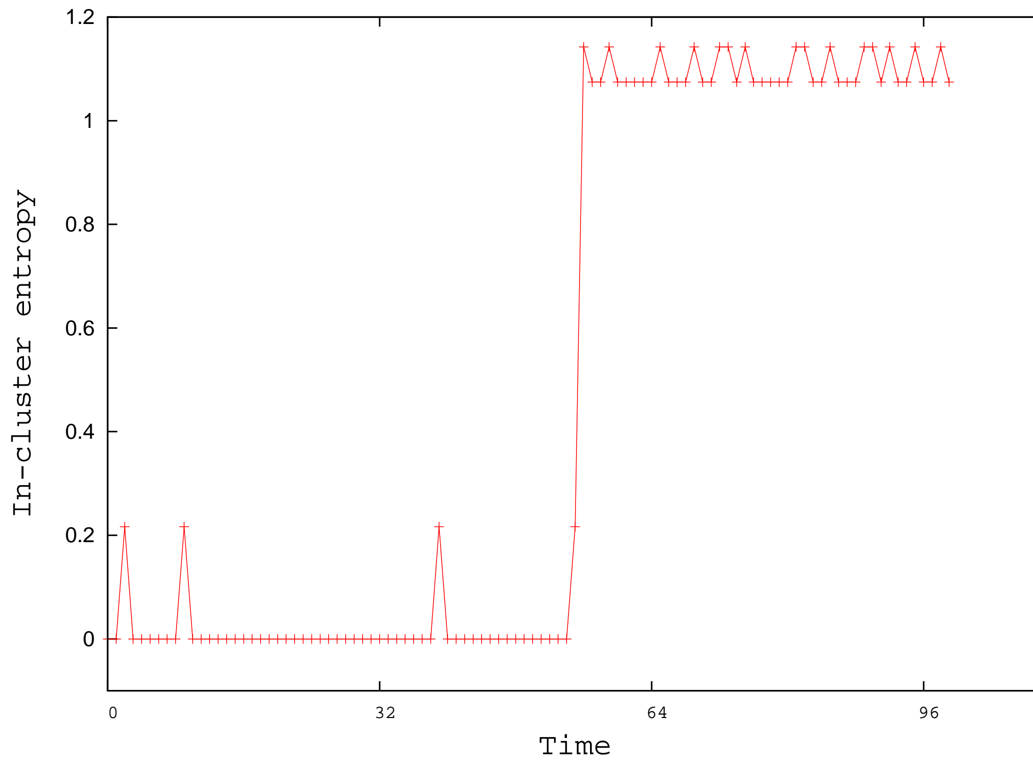


Figure 8.14: Sample in-cluster entropy

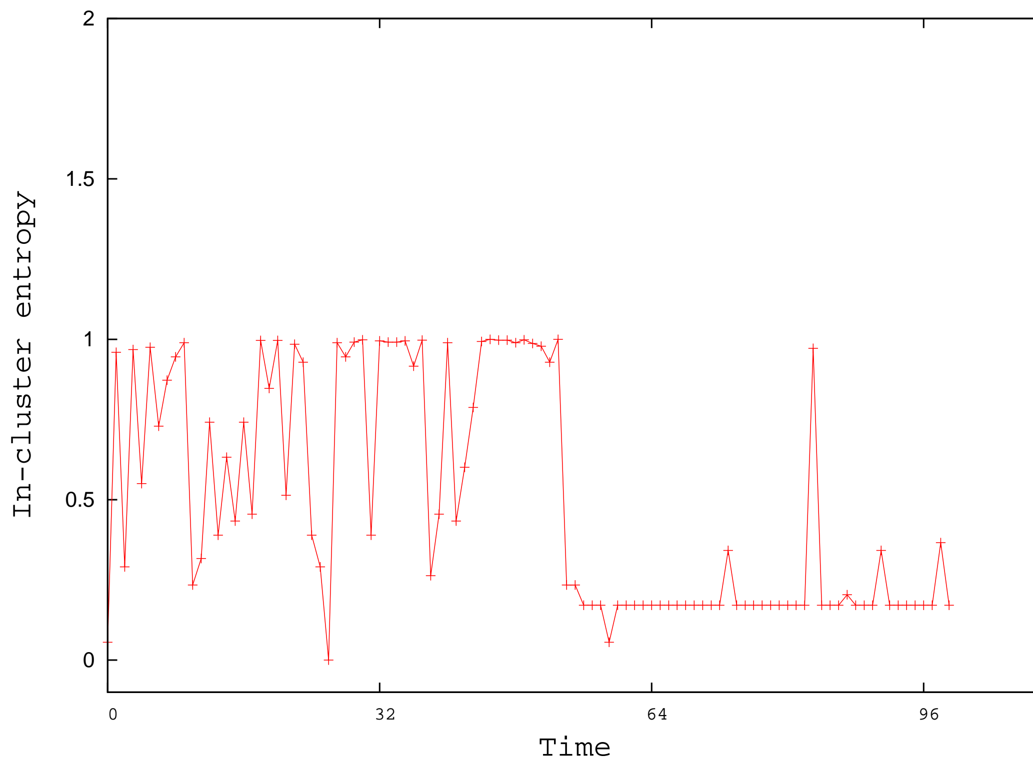


Figure 8.15: Sample in-cluster entropy

	Error detection time
Mis-ds-authentication	65
Mis-ds-connection-pool	65
Del-AccountJSP	64
Del-DisplayQuoteJSP	62
Del-TradeHomeJSP	64
Del-MarketSummaryJSP	65
Del-OrderJSP	X
Del-PortfolioJSP	X
Del-QuoteJSP	65
Mis-ThreadPool	65
DB-QuoteEJB	63
DB-HoldingEJB	60
DB-OrderEJB	64
DB-AccountEJB	65
DB-AccountProfileEJB	65
Exception-QuoteEJB	64
Exception-OrderEJB	64
Exception-HoldingEJB	61
Exception-AccountProfileEJB	64
Exception-AccountEJB	64
Exception-MarketSummaryJSP	X
Exception-QuoteJSP	66
Exception-PortfolioJSP	62
Exception-WelcomeJSP	X
Exception-AccountJSP	X
Exception-OrderJSP	X
Exception-TradeHomeJSP	X
Thread-QuoteEJB	58
Thread-OrderEJB	65
Thread-HoldingEJB	63
Thread-AccountProfileEJB	65
Thread-AccountEJB	65
Thread-MarketSummaryJSP	65
Thread-QuoteJSP	62
Thread-PortfolioJSP	65
Thread-WelcomeJSP	65
Thread-AccountJSP	64
Thread-OrderJSP	64
Thread-TradeHomeJSP	64

Table 8.9: Error detection with information-theoretic models

NMI threshold	0.99	0.875	0.75	0.625	0.5	0.375	0.25	0.125	0
True Positive	0	0	1	10	32	32	32	26	12
False Positive	0	0	0	0	1	0	0	0	0
False Negative	39	39	38	29	6	7	7	13	27
Recall	0	0	3%	25%	82%	82%	82%	66%	30%
False Positive Error Rate	0	0	0	0	3%	0	0	0	0
F-measure	0	0	0.06	0.4	0.88	0.9	0.9	0.8	0.46

Table 8.10: Error detection with different NMI thresholds

We would like to study the performance of our algorithm with different NMI threshold. There are two reasons for such a study.

First, we want to see if our algorithm is sensitive to the choice of NMI threshold. If it works well in a relatively wide range of NMI, then it would not be difficult to choose a NMI threshold for the algorithm to work.

Second, we also would like to confirm the necessity to cluster correlated metrics. Assume we set the NMI threshold to 0, then all metrics will be in the same cluster. In other words, if we do not separate the metrics into clusters based on correlation but still calculate the cluster entropy for this big cluster, will the algorithm still be effective?

Therefore, we would like to evaluate what is the overall tendency of our algorithm's performance when the NMI threshold varies.

Table 8.10 shows the impact of using different values for the NMI threshold. Nine NMI thresholds are studied. Since different NMI may result in different clusters, to ensure fairness of comparison, we monitor the five biggest clusters, which in most case include clusters with size eight or more.

We first study the two extreme cases. When the NMI threshold is set to 0, it is the case that any two metrics are considered correlated, thus there will be a single, big cluster containing all the metrics. In this case, we are simply trying to calculate the system-wide entropy. Not surprisingly, there is a lack of sensitivity and only 12 faults are detected. In the reverse case, where the NMI threshold is set to 0.01, only almost deterministic metrics are considered correlated, thus no meaningful clusters are found and we do not detect any fault.

When the NMI threshold is too low, metrics with very weak correlations are considered correlated. When the NMI threshold is too high, only very few metrics which are very strongly correlated are included. In both case the correlation threshold is not proper and the error detection results is not ideal.

The best scenario is the case when NMI thresholds lies around 0.25 to 0.5. In

this range, most faults are detected with very low false positive rate. Therefore our algorithm is not sensitive to the choice of NMI thresholds. Why 0.25 to 0.5 seem to be the best? As illustrated in section 8.3.1, NMI from 0.25 to 0.5 largely overlaps with R-square from 0.6 to 0.9 in the linear case, a range known to indicate good linear correlations. In addition, as shown in Figure 8.8, there are a number of non-linear correlations with NMI close to 0.4. As a result, an NMI threshold around this level will likely include these non-linear correlations.

8.4 Computational Cost

In this section we evaluate the computational cost of our algorithms.

The cost of the monitoring system consist of three parts: the cost to collect each metric on the local machine; the cost to transmit metric values to the machine to process the metrics; and the cost of the algorithms to process the collected metrics.

The first two costs are dependent on the specific system being studied. For the specific experimental testbed we used, there was metric collection overhead analysis and evaluation in the previous work by Munawar [65]. The performance of the system we studied is reduced by up to 12%, which is acceptable from our perspective. However, we would like to note that the metric collection cost would vary significantly with different systems. The cost to collect metrics on local machines is dependent on the hardware capability of the machine, the system and application running on the machine, and the efficiency of the tools to collect the metrics. The cost to transmit the metric values for processing is dependent on the network topology of the monitored system, as well as the way of processing metrics. For example, the metric processing could be either distributed or centralized, depending on the size of the system monitored. In our case, we have a centralized server to process the metrics. One sample consist of less than 400 metrics, each of which is up to 4 bytes. Therefore, the sample size is less than 1600 bytes. On a 100Mbps Ethernet the time to transmit one sample is less than 1 second. Therefore, for our testbed the metrics collection and transmission cost are both acceptable. A general analysis of metrics collecting cost is out of the scope of our thesis. However, the basic assumption of our work is that metrics can be collected with acceptable cost from the system, and our work starts with the metrics provided by some metric-collection framework.

We then focus on the study of the cost of processing the metrics. We carry out our algorithms on a commodity machine with an Intel Xeon 3.2GHz CPU, 4GB

	Previous OLS	Linear	Information-theoretic
Theoretic complexity	$O(m)$	$O(m)$	$O(n)$
Parameter in evaluation	$m \approx 10000$	$m \approx 4000$	$n \approx 300$
Process time for 100 samples	$2300 \sim 2500$ ms	$450 \sim 550$ ms	$13 \sim 18$ ms
Process time each sample	$23 \sim 25$ ms	$4.5 \sim 5.5$ ms	$0.13 \sim 0.18$ ms

Table 8.11: Computational cost

of memory and CentOS 3. We calculate the time elapsed for the error detection algorithms, starting from the point that all metric data has been stored in the memory. We are provided with approximately 300 metrics in our evaluation, and at each time interval a sample of these 300 metrics is processed. Since the time to process one sample is very short and the measurement of such a time cannot be accurate, we record the time to process 100 samples with the three algorithms being studied: the linear modeling, the information-theoretic modeling, and the previous OLS modeling(see Section 8.2.4).

The evaluation and analysis is reported in Table 8.11, where n is the number of metrics, m is the number of models discovered. Usually $m \approx O(n^2)$ for pairwise modeling. It can be inferred that the time to process one linear model is approximately 2 microsecond. The difference between the linear modeling and previous simple linear modeling is caused by the difference of number of models they used. The cost of the two algorithms are the same in the big O notation, and the linear solution reduces the constant by half.

The difference between the linear modeling and the information-theoretic modeling is significant, as this is a difference between $O(n)$ and $O(n^2)$ algorithms. In our evaluation, the information-theoretic solution is about 2 orders of magnitude faster compared with the linear solutions when the number of metrics is about 300.

In general , our algorithms are cost-efficient as they both take less than 10 ms to process one new sample, which is collected every 10 seconds in our evaluation. Assuming our algorithms are applied to a larger system with 10000 metrics involved. Based on the analysis the cost of the the linear modeling grows as $O(n^2)$ so it can be estimated that processing one sample should take approximately 5 second, and the information-theoretic modeling should take only approximately 5 ms. The cost of the linear modeling is still acceptable, since with a large system of 10000 metrics we can expect the computational power assigned to process the metrics would be more than the single 3.2GHz machine we used. However, the information-theoretic modeling is much more efficient, and the computational cost could be kept very low. Moreover, there is unlikely to be any algorithm much more efficient compared

with the information-theoretic modeling, because reading the n metrics once each would be an $O(n)$ algorithm. Therefore, the information-theoretic solution can easily fulfill the scalability requirement in a cloud computing environment.

Chapter 9

Conclusion and Future Work

In this dissertation we tackled the challenge of monitoring complex software systems in an automated and cost-effective manner. After surveying on related work, we abstracted a solution framework of four steps based on modeling of management metrics. Approaches following our solution framework entails modeling and monitoring complex software systems using efficient mathematical models with only metric data, and do not need to use domain knowledge, detailed information about system structure and mechanism, or prior knowledge of faults. Therefore, these approaches are widely applicable to many different systems as long as management metrics are collectible.

Incorporating two different mathematical modeling techniques with the solution framework, we devised two practical solutions to achieve automatic monitoring of the software systems. These approaches can be implemented easily and deployed with little or no change to the target systems.

Our first technique start with analyzing several common factors that reduce the effectiveness of metric-correlation models in monitoring complex software systems. We designed methods to capture these factors in the metric-correlation models. These methods include employing GLS regression and modeling multi-variable correlations and varying coefficients. We employ a non-parametric technique to detect errors by identifying significant shifts in the number of correlation models reporting outliers.

We use a realistic enterprise software system to demonstrate that our approach can successfully detect errors. The OLS and GLS regression has proven valuable in significant coverage improvements; the improvements from multi-variable models

and RLS models are less significant, which is discuss in Appendix A. As a whole, our approach is very effective in detecting most errors with a very low false-positive rate.

We devised a second technique, built on normalized mutual information, to automatically monitor the health of complex software systems and localize faulty components when errors occur. This approach consists of tracking the entropy of metric clusters. We employ the Wilcoxon Rank-Sum test to automatically identify significant changes in cluster entropy, thereby enabling robust error detection.

We evaluate the information theoretic solution using the same testbed and data we evaluate the linear modeling solution. We show through experiments that both techniques have high fault coverage and low false-alarm rate. In addition, the information theoretic solution is very computationally efficient compared with the linear-model solution.

9.1 Future Research Work

There are many future works associated with the work presented in this dissertation. Since the error detection is done by modeling the management metrics, the next challenge is to diagnose the faults in the system. The current work based on metric correlations is subject to intrinsic limitations as discussed in Appendix B. Methods to improve the fault localization are desired. Without any knowledge of domain knowledge or detailed information about system structure and mechanism, it may be very difficult to do diagnosis. Therefore, an interesting study may start with the study of what is a reasonable expectation of additional knowledge or information we can take into consideration when try to do diagnosis.

Although our solutions are very efficient given the fixed number of metrics, possible improvements are possible by trying to minimize the number of metrics being modeled. It is possible that for very large distributed systems we have a large number of metrics available. In such cases, pre-selection of the metrics to reduce the size of modeled metrics could be a promising direction to reduce monitoring overhead.

One other promising approach is to assume more knowledge about the faults instead of assuming more knowledge about the systems. By assuming knowledge of previous faults there has been a lot of related work which yields accurate fault diagnosis. This could be another potential research area. We have previously done some preliminary work in this area [36].

After fault localization or diagnosis, the next interesting problem is to put the system back to a healthy state. This is even more challenging. Some existing work assumes there are a number of recovery actions that the monitoring system can automatically perform. Therefore, according to the metrics monitored, a preferred action could be suggested automatically with the monitoring system.

If the error detection, fault diagnosis and error recovery may all be done effectively in a coherent framework, we can achieve the goal of self-managing systems and make software systems much more reliable and reduce a large portion of human workload. However, those problems are very challenging and we are expecting the researchers working on these problems for years in the future. In sum, the progress and lessons learned in this dissertation, could be considered as a few first steps contributing to the goal of self-managing systems.

Appendix A

Addressing Specific Problems in Linear Modeling

We identified a few problems of OLS linear models in Section 6.1. We addressed the major problem of heteroscedasticity in Section 6.2. We also made an attempt to address the two other minor problems specifically: missing variable and varying coefficients. The methods and evaluation are presented in this appendix.

A.1 Modeling Varying Coefficients

We developed a method to explicitly handle the problem of varying coefficients identified in Section 6.1.2.

If the coefficients of a regression model evolve, then we need to ensure that the model is up-to-date; otherwise, analysis based on such models may be misleading. Therefore, every new sample, provided it is not an outlier, should be included in the regression computation to keep the coefficients current. However, simply relearning the models at the arrival of every new sample would be too costly. To maintain the advantage of the low computational cost of OLS regression, we use a recursive method to update the model when new a sample arrives.

The recursive algorithm is well studied. The formula to update the model with new sample $(y, \mathbf{x})$ is given by:

$$\mathbf{k} = \frac{1}{1 + \mathbf{x}'\mathbf{P_0}\mathbf{x}}\mathbf{P_0}\mathbf{x} \tag{A.1}$$

$$\beta = \beta_{\mathbf{0}} + \mathbf{k}(y - \mathbf{x}'\beta_{\mathbf{0}}) \tag{A.2}$$

$$\mathbf{P} = (\mathbf{I} - \mathbf{k}\mathbf{x}')\mathbf{P_0} \tag{A.3}$$

where P_0 may be initialized with $P_0 = (X_0' X_0)^{-1}$.

A.2 Modeling Three-Variable Correlation

We developed another method to explicitly handle the problem of missing variables identified in Section 6.1.3.

When we try to model a multi-variable correlation using a two-variable model, we are likely to see non-constant residual variance because of the missing variable(s). Specifically, when the true relationship is $y_i = \beta_0 + \beta_1 x_{1i} + \beta_2 x_{2i} + e_i$, a model of the form $y_i = \beta_0 + \beta_1 x_{1i} + e'_i$ will yield the residuals $e'_i = \beta_2 z_i + e_i$, which do not have constant variance. As such, we can use non-constant error variance as a clue in finding multi-variable correlation.

Our greedy algorithm shown in Algorithm 1 builds on this insight to find models with missing variables .

An exhaustive search of three-variable models cost $O(n^3 C)$, where n is the number of metrics, and C is the cost to generate a model with given metrics. Our algorithm, on the other hand, costs only $O(gnC)$, where g is the number of GLS models, which is less than $\frac{n^2}{2}$ in the worst case. Therefore, our algorithm cost no more than an exhaustive search. In addition, if g is much fewer than n^2, the algorithm could be much more efficient than an exhaustive search.

A.3 System Monitoring with New Models

We have four types of models to quantify linear relationships between metrics in a system. We make small modifications to the solution we presented in Section 6.3 to detect errors in the system with the four types of models. The new procedure is as follows:

Algorithm 1: Algorithm to identify three-variable models efficiently

```
Input: 𝔾 ;                                        // set of GLS models
𝕆 ;                                                // set of OLS models
𝕄 ;                                                // set of all metrics
Output: 𝕋 ;                                        // set of three-variable models
begin
    𝕋 := ∅
    foreach g ∈ 𝔾 do
        Find metrics m1 and m2 modeled by g
        find S1 = {m|m ∈ 𝕄, m and m1 related via a model in O}
        find S2 = {m|m ∈ 𝕄, m and m2 related via a model in O}
        foreach m ∈ 𝕄 \ S1 \ S2 do
            Learn the three-variable model t with m1,m2 and m and compute
            the F-score f
            if f > F_{2,n-3,α} then
                𝕋 := 𝕋 + {t}
end
```

1. *Metric Modeling*: We learn metric correlation models based on metric samples collected during a normal running period. This is usually done offline so the models are prepared before we start monitoring the target system.

2. *System Health Signature Generating*: For each type of models we found, we use the outlier count w_t (defined in section 5.3) as the system health signature. This signature is generated online as we collect real time samples from the monitored system.

3. *System State Checking*: We consider persistent changes in the system health signature as an indication of existence of errors. Therefore, we use Wilcoxon Rank-Sum test to detect persistent changes in the outlier count for each type of models to do system state checking.

We may end up with several types of models for the system in the metric modeling step (*e.g.*, both ordinary least square models and generalized least square models are found). For each type of models, we apply the three-step procedure and raise alarms separately. We considered an error is detected when any type of models raise an alarm.

The approach is illustrated in in Fig A.1. The metric modeling step is discussed in Section A.3.1 and the next two steps are similar to those discussed in Section 6.3.

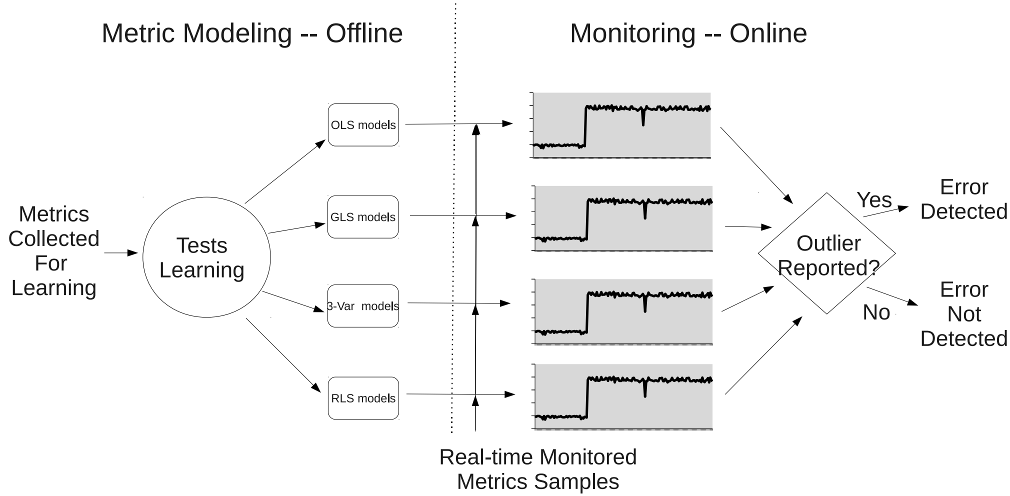


Figure A.1: Model learning and system monitoring

A.3.1 Metric Modeling

Figure A.2 presents our approach to identifying the appropriate modeling technique. In the figure *pass* means that we accept the null hypothesis (*i.e.*, the error variance is constant). For each pair of metrics, we first employ the White test to see if the residual variance of two-variable linear regression models is constant. Next, the Goldfeld-Quandt test is used to check whether the GLS models can capture the observed heteroscedasticity. If both tests suggest that the residual variance is constant, we model the relationship using OLS regression. If both the White and the Goldfeld-Quandt tests fail, which suggests that the use of GLS regression may be appropriate, we employ the GLS model. If the White test suggests heteroscedasticity, but the Goldfeld-Quandt test does not, we search for three-variable models. This procedure gives us two categories of models: GLS models, and the three-variable models. Finally, since we do not have ways to directly test for varying coefficients, we then employ OLS regression to each pair of metrics and find those with high fitness score. In addition, if the pair of metrics also fail the White test, then varying coefficient is possible, and we use RLS to update the model during monitoring.

All model learning are done offline with one exception: the RLS models, which are actually good OLS models but we suspect that they may have varying coefficients. For those models, we need to update the model at every new sample in order to be updated with varying coefficients. Fortunately, recursive least square is efficient so it may be implemented online. For those RLS models, we do not trust

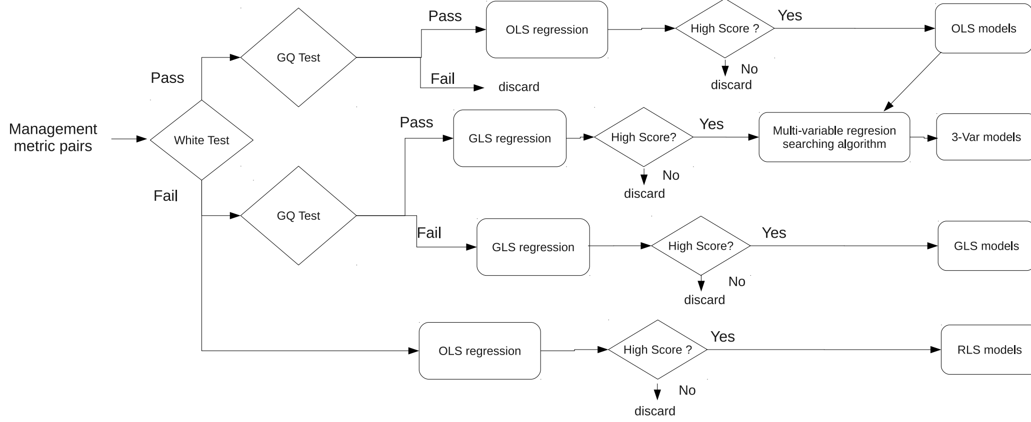


Figure A.2: Learning metric correlation models

the original coefficients we learned. Therefore at the beginning of the monitoring we assume the system is in a normal state for a short time, and use samples collected during that time to reconstruct the model, then update the model at the arriving of each new sample.

A.4 Evaluation

We use the same testbed as described in Section 8.1 to evaluate the method. Our learning procedure results in 988 OLS models, 3219 GLS models, 5533 multi-variable models, and 10501 RLS models. We first show a few examples where faults are detected.

Figure A.3 shows the number of RLS models that report outliers during one of the experiments. For RLS, we only use knowledge of which metrics are correlated and re-learn the model parameters. As such, we use samples before time 20 to estimate the model parameters, and we start testing new samples at time 20. There is almost no model reporting outliers at the beginning, but at time 56, when a fault is injected, we see more than 800 models suddenly reporting outliers. Since we keep updating the parameters of the models, after a few more samples the models adjust to the anomalous behavior and stop reporting outliers. However, we can see that it is easy to infer that the system state changed at time 56.

Figure A.4 shows the number of three-variable models that report outliers for the same experiment. We see that approximately 300 models (or 5% of the 5533

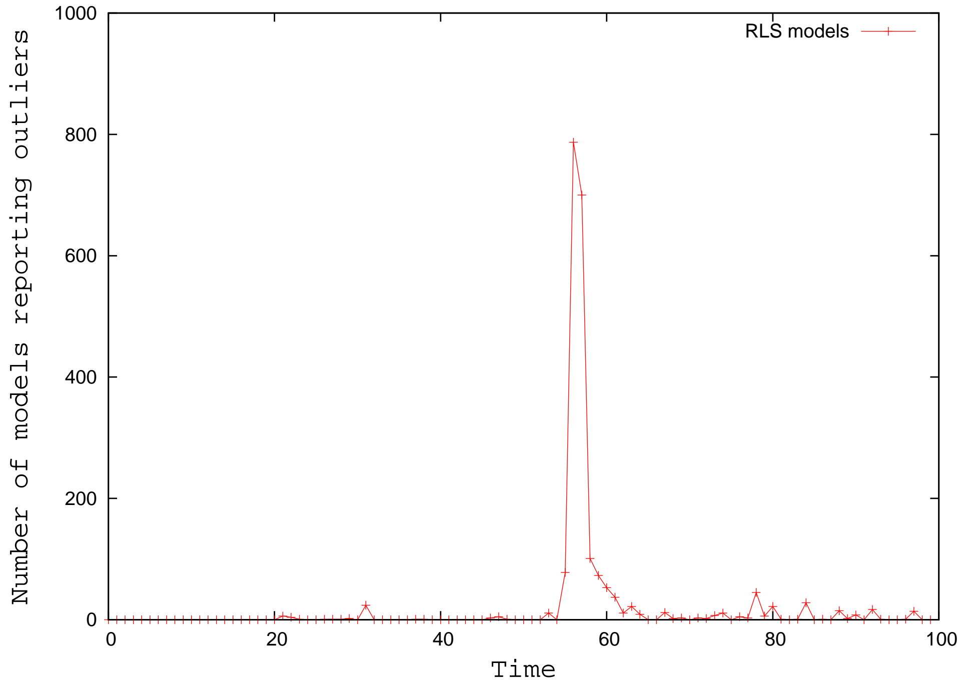


Figure A.3: Sample fault detection - a simple case

multi-variable models) do not hold from the very beginning. Still, we see evidence of system anomaly at time 56, when the number of broken models increase to more than 400.

In Figure A.5 we show two examples of fault-injection experiments and the results of our monitoring. We plot the number of models that report outliers at each sampling period for the four categories of models. The time axis starts at offset 0, and thus the fault is injected at time-interval 56.

In the first example, shown in Figure A.5(a), we observe a significant change in the number of models reporting outliers for all four types of models soon after the fault is injected. In the second example, shown in Figure A.5(b), with a different fault injected, we observe that the three-variable models fail to detect the fault. These examples indicate that different categories of models may perform better in detecting errors caused by different faults. This is partly explained by the different coverage of metrics of the different categories of models. These examples also suggest that we could improve error detection by raising an alarm when models from any of the four categories report outliers, provided doing so does not increase the false-positive rate.

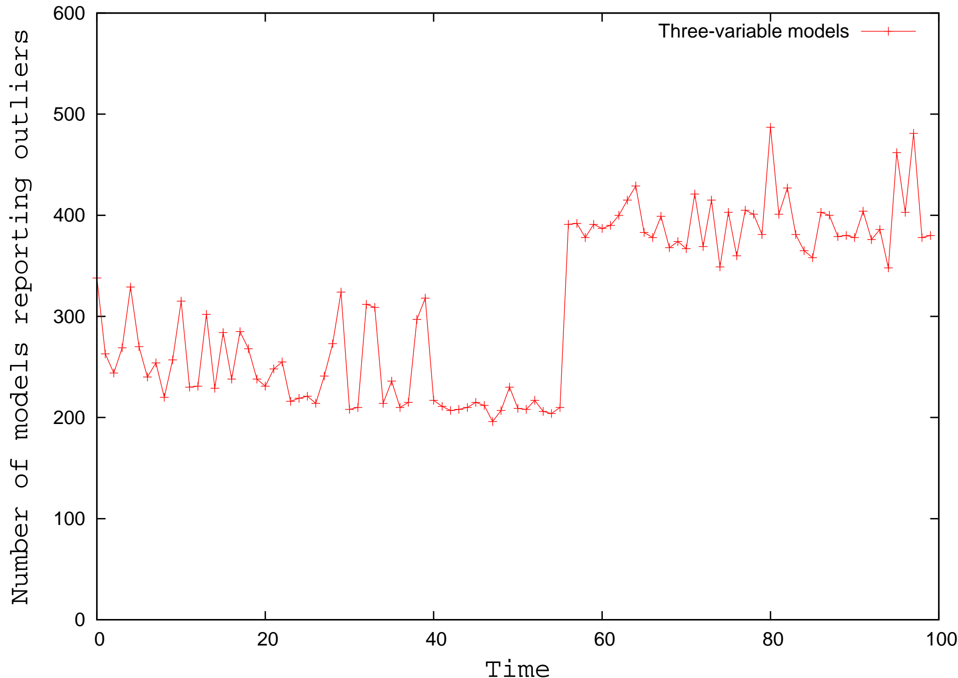


Figure A.4: Sample fault detection - tolerating invalid models

A.4.1 Error-Detection Results

We present results of error detection in our 39 fault-injection experiments in Table A.1. In the first four columns we show results for all four categories of models. Any alarm raised by a method within the Wilcoxon Rank-Sum test window from the point at which fault is injected is considered a successful detection. Errors reported before the point at which the fault is injected is considered a false positive. If a fault is detected by only one of the category of models, we count it as a "unique contribution" in the table.

The first observation is that RLS models less reliable than models in the other categories. With RLS models we have three fault-injection experiments with false alarms, while with models from the other categories, we have no false alarms. In addition, RLS models have no unique contribution, and they detect the least number of faults among all four categories. Therefore, we do not use RLS in subsequent analysis.

We combine the results of the other three categories of models by declaring errors in the system when any of the three categories report outliers. The results are shown under the heading "combined" in Table A.1. We can see that such a combination enables us to detect 34 out of 39 faults without false alarms. We

	OLS	GLS	3-var	RLS	Combined
Faults detected	31	24	27	19	34
Faults with no alarm	8	15	12	17	5
Faults with false alarms	0	0	0	3	0
Unique contribution	3	1	1	0	Not defined

Table A.1: Error-detection summary

detect the one more faults compared with the current techniques. The gain is subtle considering the significant more number of models involved. Therefore, we think the OLS and GLS models are the better modeling techniques in the balance of effectiveness and efficiency.

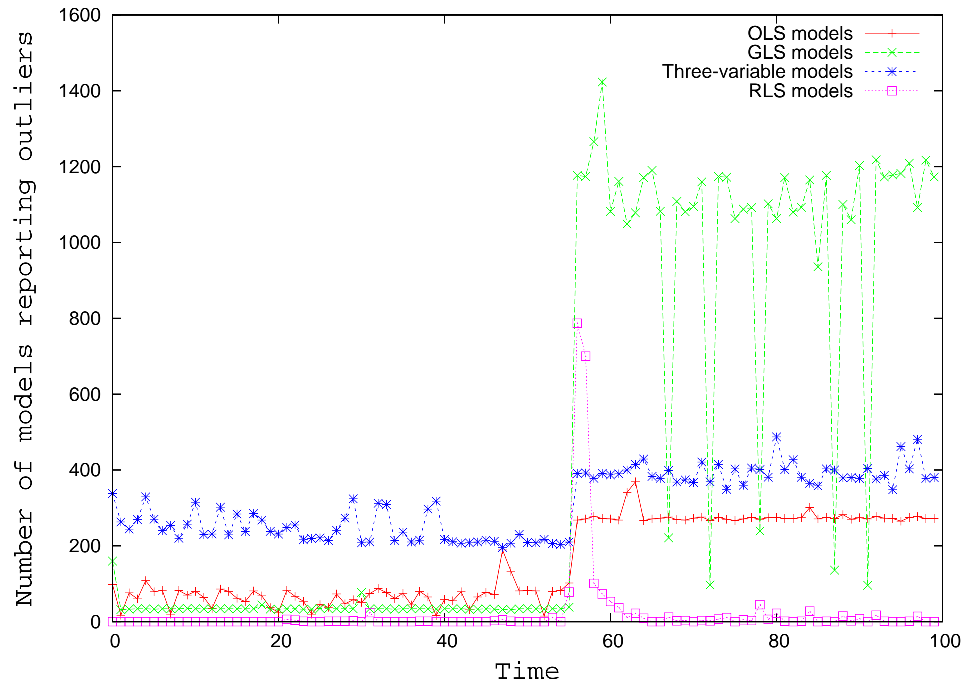
A.4.2 Understanding RLS Performance

We would like to understand why RLS models do not perform as well as the other models. Two problems plague RLS models: they detect errors in fewer fault-injection experiments and they cause more false alarms.

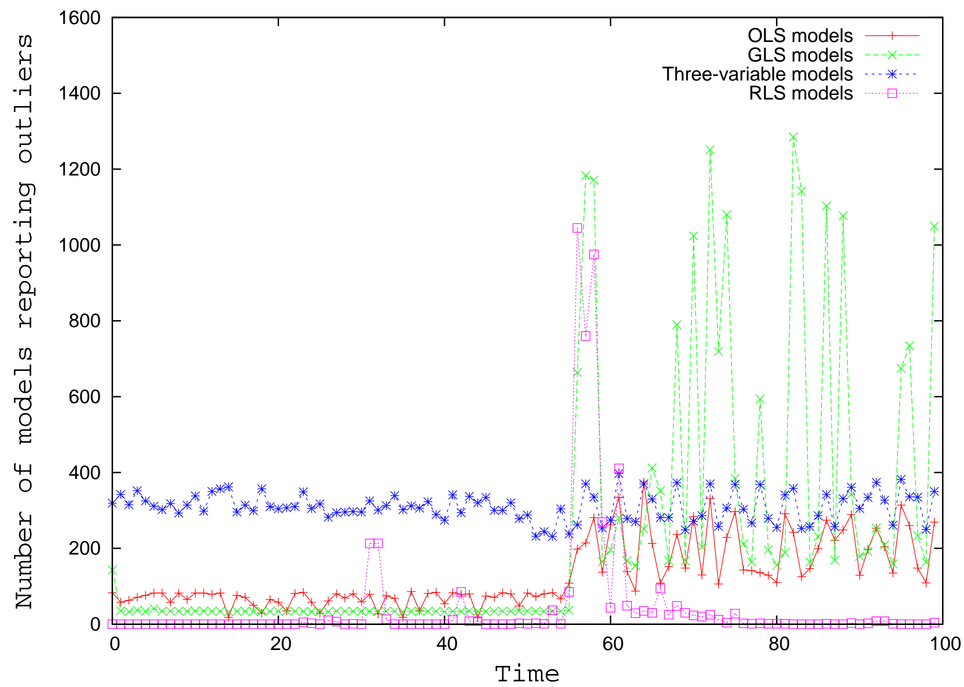
For the first problem, our analysis indicates that in many cases when RLS models miss a fault, it is not the case that none of the RLS models report outliers; instead, some outliers are reported in the beginning of the fault injection period. But RLS models update the parameters fast and thus quickly adjust to the anomalous behavior, which causes them to stop detecting outliers. This short burst of outliers is filtered by our Wilcoxon Rank-Sum test, so no alarm is ever raised.

For the second problem, our analysis shows that because of the adaptive nature of RLS models, the data fed to the Wilcoxon Rank-Sum test has very low variance, which in turn makes it sensitive to noise. To illustrate this, we consider one of our fault-injection experiments in which errors were only detected by three-variable models. Both OLS and RLS models did not detect errors. While OLS models did not report a false alarm, RLS models did. In Figure A.6 we can see that OLS models are noisy. Around 50 to 100 models report outliers throughout the experiment, and errors are not detected and no false alarms are raised. However, RLS are "noise-free" in the beginning but a small number of models (20 out of 10501) report some outliers which cause the Wilcoxon Rank-Sum test to trigger. 20 out of 10501 is a tiny model-level false positive rate, which is hard to avoid. However, because of the absence of noise in the data processed by the Wilcoxon Rank-Sum test prior to this tiny burst, a false alarm is raised. We have found that a small level of background "noise" is beneficial for enabling the monitoring system to tolerate false positives.

This also explains why the other three categories of models successfully avoided false alarms.



(a) delete-displayquote.jsp



(b) exceptions-AccountEJB

Figure A.5: Sample error-detection results

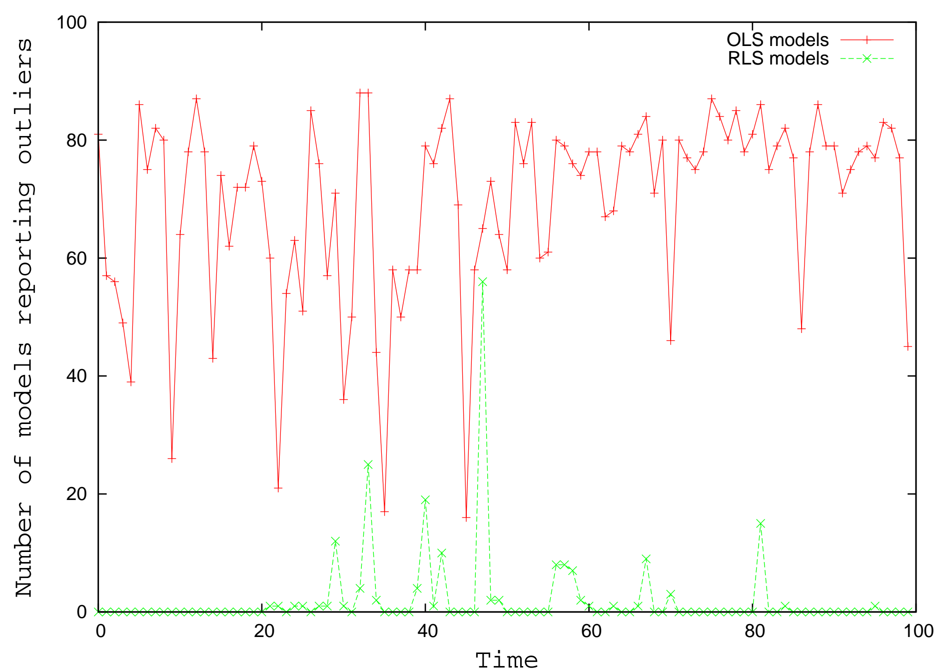


Figure A.6: OLS models *vs.* RLS models

Appendix B

Limitations of Fault Localization with Metric Correlations

In prior work it is often implicitly assumed that if a modeling technique works well in detecting anomalies in a system, then it is likely that such modeling will help in fault diagnosis. If the diagnosis does not work well, then improving the technique used to model the metrics correlation would help. However, this may not true. With only correlation information and without metric-component knowledge, it may not be feasible to make an accurate diagnosis irrespective of how effective the correlation models are in detecting errors in the system. This is the case because of some intrinsic limitations in the use of correlation models for fault localization.

Specifically, it is usually assumed that a faulty component is likely to cause many correlations involving the component's metrics to break. This assumption is somehow problematic; indeed, a faulty component is likely to cause many correlations to break, however, it is possible that a majority of the broken correlations are not associated with the faulty component. We next show an example of this phenomenon and explain it in Section B.1 and Section B.2.

Consider the example in Figure B.1. Assume there are five components A, B, C, D and E. A transaction in the system starts in component A, which calls some function in component B, which in turn calls some function in component C, *etc.* Such a dependency chains are very common in software systems.

Let t_a, t_b, t_c, t_d, t_e be the response time of components A, B, C, D and E, respectively. Let t_{ij}, $1 \leq i \leq 4$, $1 \leq j \leq 2$ be the execution time in these components as shown in Figure B.1, we have four basic relationships: $t_a = t_b + t_{11} + t_{12}$,$t_b = t_c + t_{21} + t_{22}$,$t_c = t_d + t_{31} + t_{32}$,$t_d = t_e + t_{41} + t_{42}$. Further, assume that the bot-

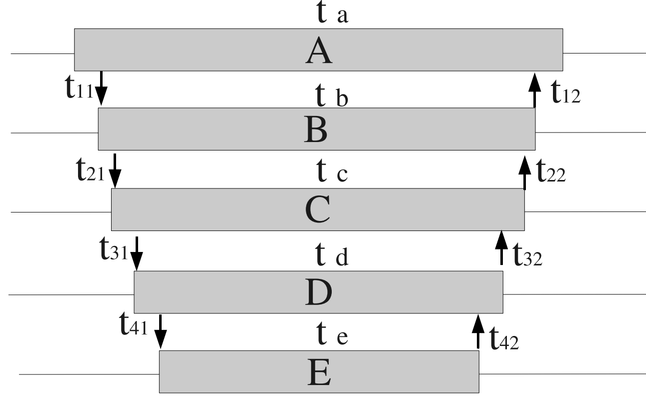


Figure B.1: Fault localization example

tleneck of the transaction is in component E, such that the transaction takes most of time in component E, *i.e.*, $t_e \gg t_{ij}$, for any i, j. As a result, when we try to discover relationships among the five metrics, we will find linear correlations such as $t_a = t_b$. In fact, since every metric should be roughly equal to t_e, every two metrics should roughly equal each other. Therefore, we end up with $C(5, 2) = 10$ correlations , where $C(n, k)$ is the number of k-combination from a set of size n, given by $C(n, k) = \frac{n!}{k!(n-k)!}$.

Now assume there is some problem with component C. As a result, it spends much more time in component C, *i.e.*, t_{31} and t_{32} become larger and are no longer negligible. In this case, the following four relationships are still valid: $t_a = t_b, t_b = t_c, t_a = t_c, t_d = t_e$, while the other six relationships no longer hold.

Because six out of ten relationships (or equivalently the corresponding models) become invalid, we can easily infer that something is wrong in the system. However, what we observe in this example clearly violates the assumption that "Correlations involving metrics from a faulty component are more likely to break when a fault is present". In this example, the faulty component is C, but metrics t_d and t_e seem to be more likely to be linked with the fault. For example, let's compare t_c and t_d: there were four correlations involving d before the fault occurred: $t_a = t_d, t_b = t_d, t_c = t_d, t_d = t_e$ However, after the fault in component C, only one correlation $t_d = t_e$ remains valid. On the other hand, there were four correlations involving c before the fault: $t_a = t_c, t_b = t_c, t_c = t_d, t_c = t_e$. After the fault, two correlations remain valid: $t_a = t_c$ and $t_b = t_c$. Clearly, component D appears to be more likely

faulty.

In fact, if we calculate the Jaccard coefficient, we have $J_a = J_b = J_c = \frac{2}{8}$ and $J_d = J_e = \frac{3}{7}$. The Jaccard coefficient-based anomaly score of C is one of the lowest among all five components. Basically, this happens because of the "spread" of correlations. In Section B.1 and Section B.2 we explain this phenomenon in a general model.

We should note that the example is not the worst scenario. In real systems, there are thousands of metrics available. Due to computational cost constrains, it may not be possible to collect all metrics. As a result, many metrics could be left out. For example, if we only collect metrics t_a and t_e in the above scenario, we will be able to discover the correlation $t_a = t_e$ during model learning and find this equation to become invalid when a fault occurs in component C. However, there is no way to know that the fault is neither in component A nor component E but in component C.

Therefore, diagnosis with only metrics and their correlations is hard, even though they work well in detecting errors in a system. Considering the prevalence of dependencies in software systems, the example is representative and reveals an intrinsic limitation of diagnosis with metric correlation models.

B.1 Simplified View of a System of Correlations

In this section we present a simplified, abstract view of the system to study how correlation between metrics exist and evolve in a software system. First, it is important to distinguish correlation from causation.

B.1.1 Causation and Correlation

There are at least two ways correlation between metrics arise. First, two metrics can show correlation if there is causality relationship between the two metrics. One example of such correlation are the response times in dependent components, as shown in Figure B.1. The response time t_b is part of response time t_a, therefore there are correlations between the two response times.

The second way for two metrics can be correlated is that they are both affected by the same third unknown factor. For example, number of visits to two different databases are both affected by the workload. Therefore, even if there is no directly

causality relationship between the two databases, the two metrics may show a correlation because they are both determined by the workload.

B.1.2 Cluster of Correlations

We use a graph $G = (V, E)$ to show the causalities between metrics in a software system. Every metrics is represented by a node $m_i \in V$. Every edge $\{m_i, m_j\} \in E$ represents a metric correlation between m_i and m_j caused by causality in the first way described in section B.1.1, where m_i is the cause of m_j. Similarly, we use a graph $G' = (V, E')$ to show the correlations between metrics in a software system. Every edge $\{m_i, m_j\} \in E$ represents a correlation between metrics m_i and m_j. We call G the causality graph, and G' the correlation graph.

To simplify the discussion, we make an assumption about the metric correlations caused by causalities: We assume that we take into account only the correlations which are deterministic with no random errors between two metrics. As a result, for any edge $\{m_i, m_j\} \in E$ there is a invertible function f_{ij} such that $m_j = f_{ij}(m_i)$. This assumption assures that the graph G consists of a few trees, since every metric is determined by only one other metric therefore every node in G has at most one parent. G' may be generated from G by observing the fact we summarized in assumption 1:

Assumption 1: Two metrics are correlated if they are in the same tree when represented in graph G.

Given the above assumption, we can see that the metrics in the same tree are correlated with each other. Therefore, each tree in G is a cluster in G' such that every two metrics in the same cluster are correlated. In reality, we typically do not know the causal relationships between metrics in a system. Instead, what we can observe is the correlations between metrics. Therefore, we do not have the graph G but we can observe the graph G', *i.e.*, we can see the metrics form several clusters, such that metrics within a single cluster are correlated with each other as depicted in Figure B.2.

Consequently, we can observe that a majority of the correlations are not caused by causality directly. Instead, many metrics are correlated because of the spread of correlations.

Hereby we estimate the proportion of correlation caused by causality in all the correlations we observe in a system. Let n be the total number of metrics in a system, k be the number of clusters, $n_i, 1 \leq i \leq k$ be the number of metrics in

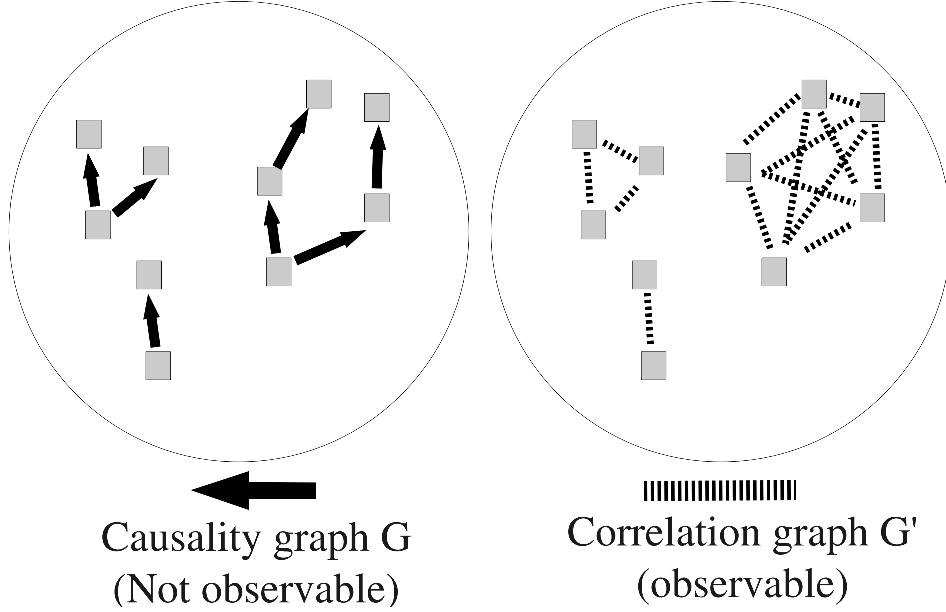


Figure B.2: Metrics' clusters

cluster i, respectively. Let A_i be the number of correlation caused by only causality in cluster i and C_i be the number of all correlations in cluster i. We have the following equations:

$$A_i = n_i - 1 \tag{B.1}$$

$$C_i = \frac{n_i!}{2!(n_i - 2)!} = \frac{n_i(n_i - 1)}{2} \tag{B.2}$$

$$n = \sum_{i=1}^{k} n_i \tag{B.3}$$

Eq. B.1 is based on the observation that each causality in cluster i corresponds to an edge in the tree i. Eq. B.2 is based on the observation that every two metrics in the same cluster are correlated. The total number of causalities, A, and the total number of correlations, C, has the ratio in Eq. B.4.

$$\frac{A}{C} = \frac{\sum_{i=1}^{k} A_i}{\sum_{i=1}^{k} C_i} = \frac{2(n - k)}{\sum_{i=1}^{k} n_i^2 - n} \tag{B.4}$$

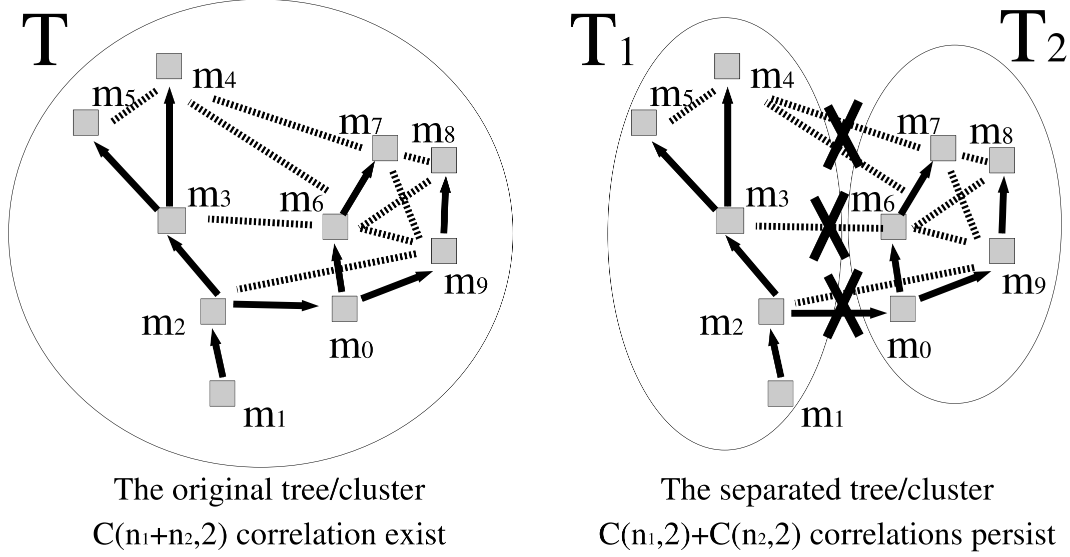


Figure B.3: The effects of invalid causality

Based on our observation from real systems, we find that usually there is a big cluster which consist of metrics correlated with the workload. Assume this is cluster 1 and $n_1 \geq \frac{n}{3}$. We have

$$\frac{A}{C} \leq \frac{2n}{n_1^2 - n} \leq \frac{18}{n - 9} \tag{B.5}$$

Therefore, if there are more than 100 metrics in the system, the proportion of correlation caused by causality would be less than 20% ($\frac{18}{91}$) of all the correlations. In other word, the majority of correlations we learn statistically does not imply causality.

B.1.3 Effects of Invalid Causality

We now estimate the effects of errors that cause some causal relationship to become invalid. Assume a causal relationship is invalid during the occurrence of some faults. In the causality graph G, an edge $\{m_i, m_j\}$ is removed. Denote the original tree that contains the edge $\{m_i, m_j\}$ as T. The removal of the edge $\{m_i, m_j\}$ will break T into two separate trees, denoted by T_1 and T_2.

While the effect of invalid causality on the causality graph G is just the removal of a single causal edge, the effect on the correlation graph G' is much more signif-

icant. Originally all metric pairs in T are correlated with each other. After the removal of the edge $\{m_i, m_j\}$, T will be broken down into two separate trees T_1 and T_2. According to Assumption 1 in section B.1.2, all metric pairs in T_1 or T_2 will remain correlated, and all metric pairs with one metric from T_1 and the other from T_2 will no longer be correlated. This is illustrated by the example in Figure B.3, where the removal of a single edge $\{m_2, m_0\}$ breaks the original tree (cluster) T into two separate trees(clusters) T_1 and T_2, and consequently many correlations other than those with m_0 or m_2 break. For example, metric pair $\{m_3, m_6\}$ are no longer correlated, neither are $\{m_4, m_7\}$. Those metric pairs in either T_1 or T_2 will persist, *e.g.*, $\{m_4, m_5\}$, $\{m_7, m_9\}$ and $\{m_6, m_8\}$.

To estimate how a fault affects the correlations in a cluster, assume there are n_1 metrics in cluster T_1, and n_2 metrics in cluster T_2. Originally, all metrics in cluster T are correlated with each other, giving us $C(n_1 + n_2, 2)$ correlations. After a fault occurs, cluster T_1 and T_2 are separated by the removal of causality edge $\{m_i, m_j\}$. The metrics within cluster T_1 are still correlated; the same applied for the metrics in cluster T_2. Therefore, a total number of $C(n_1, 2) + C(n_2, 2)$ correlations still persist. Any correlation between a metrics in cluster T_1 and a metrics in cluster T_2 becomes invalid. Therefore, a total number of $n_1 n_2$ correlations break. It is easy to confirm that $C(n_1 + n_2, 2) = C(n_1, 2) + C(n_2, 2) + n_1 n_2$.

We first evaluate how large a proportion of correlations would break. The ratio R of invalid correlation and original correlation is:

$$R = \frac{n_1 n_2}{C(n_1 + n_2, 2)} = \frac{2 n_1 n_2}{(n_1 + n_2)(n_1 + n_2 - 1)}$$

For any given tree/cluster, $n = n_1 + n_2$ is fixed. It can be proved that

$$R \leq \frac{(n_1 + n_2)^2}{2(n_1 + n_2)(n_1 + n_2 - 1)} = \frac{n^2}{2(n^2 - n)} = \frac{1}{2} + \frac{1}{2(n-1)}$$

where "=" is achieved when $n_1 = n_2 = \frac{n}{2}$. In other words, in the worst case more than half of the original correlations would break, even if most of them have no causal relationship with any of the faulty metrics (m_i or m_j). This illustrates why it is easy to detect most errors by modeling and tracking metric correlations; a small error could result in up to half correlations to break.

However, diagnosis becomes very difficult with so many correlations becoming invalid. A majority of correlations do not imply causality, and their invalidity in the occurrence of a fault does not imply any invalid causality. For example, in

Figure B.3, metrics m_3 and m_6 were correlated because they had a common factor m_2; when error occurs and affect the causality $\{m_2, m_0\}$, they become uncorrelated. However, without system specific knowledge, we do not know the causality graph but can only observe the correlation graph G'. Therefore we do not know if the breaking of the correlation between m_3 and m_6 is caused by the breaking of causality between m_2 and m_0. Thus, we cannot make a correct diagnosis.

As illustrated in section B.1.2, the majority of correlations we observed usually do not imply causality. Similarly, the majority of correlations broken during the occurrence of a fault may not imply any change in causality. This violates the usual assumption that the faulty component will mostly cause its own metrics and associated correlations to show anomalous behavior. In fact, a faulty component will cause many irrelevant metrics to show anomalies in their correlations, and thus provide unreliable information for diagnosis.

B.2 Realistic Model of a System of Correlations

In this section we extend our observation by make more realistic assumptions of the system.

To simplify the discussion, we made an assumption about the metrics correlations caused by causalities in section B.1.2: we assume we take into account only correlations which are deterministic between two metrics. This assumption inspires our assumption 1, however, the real world may be more complicated. The assumption 1 could be relaxed to model the real system better.

We observe that the relationship between metrics may not be deterministic. For a metric pair with causality, the simple and clear relationship $m_i = f(m_j)$ may not always exist. Instead, the relationship may be modeled as $m_i = f(m_j) + e$, where e is some random error. The reason is that m_j is determined by more than more factors. The true relationship may be modeled by $m_i = f(m_j, m_k, ...)$. However, m_i is the major factor that contributes most to m_j such that m_i almost determined m_j; the combination of other minor factors contribute to a small variation, which is modeled by the error e.

As a result, the Assumption 1 in section B.1.2 will become invalid. If two metrics are far from each other in the graph G, they may not be statistically strongly correlated even if they are in the same tree.

Instead of Assumption 1, we make another more realistic assumption:

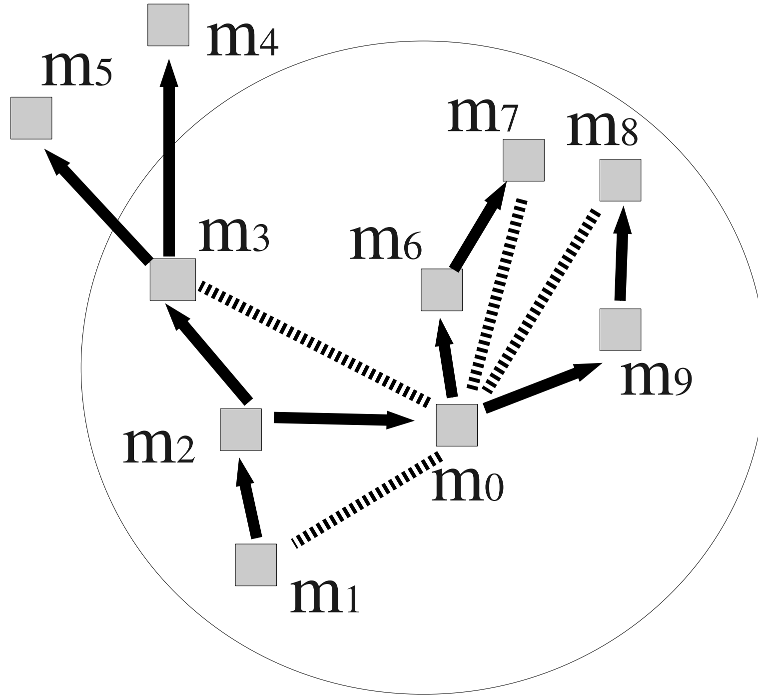


Figure B.4: Realistic view of correlations by m_0

Assumption 2: Two metrics are correlated if they are in the same tree and there is a short path in G that connects the two metrics.

We then study how the induction based on the new assumption would change to approximate the real metric correlations better.

B.2.1 Cluster of Correlations

The correlation graph and cluster of correlations would become more complicated under the realistic assumptions. Figure B.4 shows an example of the causality graph with correlations involving metrics m_0. Assume the short path in Assumption 2 is the path with a length at most two. Thus, only metrics in the same tree as m_0 and with a distance less than three are guaranteed to be correlated with m_0. In the example in Figure B.4, all metrics in the circle are correlated with m_0. m_4 and m_5 were correlated with m_0 under the simplified model (See Fig. B.2 and B.3), however, based on the realistic assumption, they are no longer guaranteed correlation with m_0.

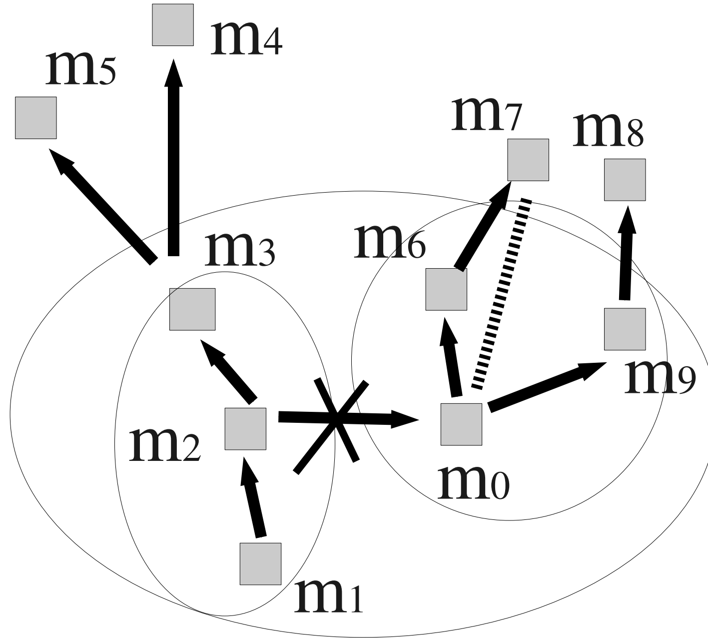


Figure B.5: Realistic view of the effects of a fault

However, even if under the realistic assumption, we can see in this example that a minority of correlations (m_2, m_6, m_9) with metrics m_0 implies causality, while a majority correlations (m_1, m_3, m_7, m_8) with metrics m_0 do not imply direct causality.

B.2.2 Effects of Invalid Causality

The effects of a fault also change from the case that we made the simplified assumption. Assume there is a fault that invalid the causality $\{m_2, m_0\}$, as depicted in Figure B.5, only correlations within the large circle are affected by the fault. All correlations involving m_4, m_5, m_7, m_8 persist, because they are too far away from the invalid causality. However, within the circle, it is the same case as we analyze in section B.1.3: the original tree are separated into two trees, and correlation within each tree persist but correlation between metrics from the two trees break. Therefore, as discussed in section B.1.3, after excluding the metrics outside of the large circle, we cannot find the invalid causality because we can only observe the invalid correlations. Consequently, we cannot localize the fault to a specific component within the large circle just by studying the change of correlations.

How does this affects the diagnosis? First of all, many correlations involving metrics which are far away from the actual invalid causality are not affected. Therefore, it is hopefully that we can exclude a number of components from suspicion since metrics in these components have all their correlations persist. Second, similar to the case we analyze in section B.1.3, correlations involving many metrics close to the invalid correlation become invalid, which may be misleading for diagnosis. In particular, we can hardly distinguish the metrics within a short distance from the metrics in the actual invalid causality, since many correlations in this range is affected. Therefore, accurate diagnosis is not possible.

Therefore, we are convinced that diagnosis with correlation models is able to exclude some components from suspicion. If we assign anomaly score to components, those components will get a lower score compared with the faulty component. However, within the left components, it is unlikely that the faulty component will be assigned a higher anomaly score compared with the others. Therefore, diagnosis with metric correlation models will be useful to reduce people's efforts to locate the fault. However, we should not expect it to be able to assign the faulty component the highest anomaly score.

Bibliographical Notes

The work presented in this thesis builds on ideas published by the author in [37, 38, 39, 40, 41, 42]. Extensions and related work to which the author has contributed includes [36, 56, 57, 58, 59, 60, 61, 62, 63, 64].

References

[1] Apache Software Foundation. Apache Module mod_status. http://httpd.-apache.org/docs/2.0/mod/mod_status.html. 14

[2] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004. 6

[3] Mark Brodie, Sheng Ma, Leonid Rachevsky, and Jon Champlin. Automated problem determination using call-stack matching. *J. Network and Systems Mgmt.*, 13(2):219–237, June 2005. 27

[4] A. Brown, G. Kar, and A. Keller. An active approach to characterizing dynamic dependencies for problem determination in a distributed environment. In *Proc. of IFIP/IEEE International Symposium on Integrated Network Management*, pages 377–390, May 2001. 25

[5] Lubomír Bulej, Tomáš Kalibera, and Petr Tma. Repeated results analysis for middleware regression benchmarking. *Perform. Eval.*, 60:345–358, May 2005. 25

[6] Andrew Byde, Dave Cliff, and Matthew Williamson. HP Labs' complex adaptive systems group research overview. Technical Report HPL-2004-79, HP Laboratories Palo Alto, 2004. 2

[7] Bryan Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic instrumentation of production systems. In *USENIX Annual Technical Conference, General Track*, pages 15–28, 2004. 13

[8] J. Case, M. Fedor, M. Schoffstall, and J. Davin. A Simple Network Management Protocol (SNMP). IETF RFC 1157. http://www.ietf.org/rfc/rfc1157.txt. 8, 9

[9] Haifeng Chen, Guofei Jiang, Cristian Ungureanu, and Kenji Yoshihira. Failure detection and localization in component based systems by online tracking. In *KDD*, pages 750–755, 2005. 27

[10] M. Chen, A. Zheng, J. Lloyd, M. Jordan, and E. Brewer. Failure diagnosis using decision trees. In *ICAC*, 2004. 26

[11] Mike Y. Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric A. Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *ICDSN*, pages 595–604, 2002. 26, 28

[12] Ira Cohen, Moises Goldszmidt, Terence Kelly, Julie Symons, and Jeff Chase. Correlating instrumentation data to system states: A building block for automated diagnosis and control. In *OSDI*, pages 231–244, 2004. 28

[13] Ira Cohen, Steve Zhang, Moisés Goldszmidt, Julie Symons, Terence Kelly, and Armando Fox. Capturing, indexing, clustering, and retrieving system history. In *SOSP*, pages 105–118, 2005. 28

[14] Joyce Coleman and Tony Lau. Set up and run a Trade6 benchmark with DB2 UDB. http://www128.ibm.com/developerworks/edu/dm-dw-dm-0506-lau.html?S_TACT=105AGX11&S_CMP=LIB. 76

[15] R. D. Cook and S. Weisberg. *Residual and Influence in Regression*. Chapman and Hall, New York, 1982. 21

[16] William H. Crown. *Statistical Models for the Social and Behavioral Sciences: Multiple Regression and Limited-Dependent Variable Models*. Greenwood Publishing Group, 1998. 55, 57

[17] Yixin Diao, Frank Eskesen, Steve Froehlich, Joseph L. Hellerstein, Alexander Keller, Lisa Spainhower, and Maheswaran Surendra. Generic on-line discovery of quantitative models for service level management. In *IM*, pages 157–170, 2003. 21, 54

[18] Armando Fox and David Patterson. Self-repairing computers. *Scientific American*, June 2003. 2

[19] Saeed Ghanbari and Cristiana Amza. Semantic-driven model composition for accurate anomaly diagnosis. *Autonomic Computing, International Conference on*, 0:35–44, 2008. 28

[20] Saeed Ghanbari and Cristiana Amza. Semantic-driven model composition for accurate anomaly diagnosis. In *International Conference on Autonomic Computing, 2008. (ICAC)*, pages 35–44, June 2008. 41

[21] Jean Dickinson Gibbons and Subhabrata Chakraborti. *Nonparametric Statistical Inference*. CRC Press, 2003. 44

[22] Zhen Guo, Guofei Jiang, Haifeng Chen, and Kenji Yoshihira. Tracking probabilistic correlation of monitoring data for fault detection in complex systems. In *ICDSN*, pages 259–268, 2006. 3, 21, 23, 24, 41, 50

[23] Jiawei Han and Micheline Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 2nd edition, 2006. 67

[24] Matthias Hauswirth, Peter F. Sweeney, Amer Diwan, and Michael Hind. Vertical profiling: Understanding the behavior of object-oriented applications. In *Proc. of 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2004. 25

[25] Daniel E. Hecker. Occupational employment projections to 2014. *Monthly Labor Review*, pages 70–101, Nov. 2005. 1

[26] IBM Corporation. Autonomic Computing. http://www.research.ibm.com/-autonomic/. 2

[27] IBM Corporation. DB2 V8.2 - System Monitor Guide and Reference. ftp://ftp.software.ibm.com/ps/products/db2/info/vr82/pdf/en_US/-db2f0e81.pdf. 13

[28] IBM Corporation. WebSphere Application Server. http://www.ibm.com/-software/webservers/appserv/. 74

[29] IBM Corporation. DB2 Universal Database. http://www.ibm.com/software/-data/db2/udb/. 74

[30] IBM Corporation. PlantsByWebSphere Sample. http://www.ibm.com/-developerworks/websphere/library/samples/plantsby.html. 75

[31] IBM Corporation. Websphere application server, version 6.0.x - monitoring overall system health. http://publib.boulder.ibm.com/infocenter/-wasinfo/v6r0/index.jsp?topic=/com.ibm.websphere.base.doc/info/aes/ae/-tprf_monitoringhealth.html. 14

[32] JBoss Enterprise. A Framework for Organizing Cross Cutting Concerns. http://jboss.org/jbossaop/. 13

[33] Guofei Jiang, Haifeng Chen, and Kenji Yoshihira. Discovering likely invariants of distributed transaction systems for autonomic system management. In *ICAC*, pages 199–208, 2006. 21, 27

[34] Guofei Jiang, Haifeng Chen, and Kenji Yoshihira. Modeling and tracking of transaction flow dynamics for fault detection in complex systems. *Trans. on Dependable and Secure Computing*, 3(4):312–326, 2006. 3, 21, 22, 23, 35, 41, 45

[35] Guofei Jiang, Haifeng Chen, and Kenji Yoshihira. Efficient and scalable algorithms for inferring likely invariants in distributed systems. *TKDE*, 19(11):1508–1523, 2007. 23

[36] Miao Jiang, Mohammad A. Munawar, Thomas Reidemeister, and Paul A. S. Ward. Detection and diagnosis of recurrent faults in software systems by invariant analysis. In *Proceedings of the IEEE High Assurance Systems Engineering Symposium (HASE)*, 2008. 28, 105, 129

[37] Miao Jiang, Mohammad A. Munawar, Thomas Reidemeister, and Paul A. S. Ward. Information-theoretic modeling for tracking the health of complex software systems. In *Proceedings of the International Conference on Computer Science and Software Engineering (CASCON)*, 2008. 50, 129

[38] Miao Jiang, Mohammad A. Munawar, Thomas Reidemeister, and Paul A. S. Ward. Automatic fault detection and diagnosis using information-theoretic modeling. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, 2009. 129

[39] Miao Jiang, Mohammad A. Munawar, Thomas Reidemeister, and Paul A. S. Ward. Heteroscedastic models to track relationships between management metrics. In *Proceedings of the International Symposium on Integrated Network Management (IM)*, 2009. 129

[40] Miao Jiang, Mohammad A. Munawar, Thomas Reidemeister, and Paul A. S. Ward. System monitoring with metric-correlation models: Problems and solutions. In *Proceedings of the International Conference on Autonomic Computing (ICAC)*, 2009. 129

[41] Miao Jiang, Mohammad A. Munawar, Thomas Reidemeister, and Paul A. S. Ward. Efficient fault detection and diagnosis in complex software systems with information-theoretic monitoring. *Dependable and Secure Computing, IEEE Transactions on*, 8(4):510 –522, july-aug. 2011. 129

[42] Miao Jiang, Mohammad A. Munawar, Thomas Reidemeister, and Paul A. S. Ward. System monitoring with metric-correlation models. *Accepted in IEEE Transactions on Network and Service Management*, 2011. 129

[43] Mark W. Johnson. Monitoring and diagnosing applications with arm 4.0. In *Proceedings of the Computer Measurement Group (CMG) Conference*, pages 473–484, 2004. 14

[44] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, 2003. 1, 2

[45] Emre Kiciman and Armando Fox. Detecting application-level failures in component-based internet services. *IEEE Transactions on Neural Networks*, 16(5):1027–1041, September 2005. 26

[46] T. Alan Lacey and Benjamin Wright. Occupational employment projections to 2018. *Monthly Labor Review*, pages 82–123, November 2009. 1

[47] Lawrence Berkeley National Laboratory. The Internet Traffic Archive. http://ita.ee.lbl.gov/html/traces.html. 74

[48] Michael R. Lyu, editor. *Handbook of software reliability and system reliability.* McGraw-Hill, Inc., Hightstown, NJ, USA, 1996. 7

[49] H. Malik, B. Adams, A.E. Hassan, P. Flora, and G. Hamann. Using load tests to automatically compare the subsystems of a large enterprise system. In *Computer Software and Applications Conference (COMPSAC), 2010 IEEE 34th Annual*, pages 117 –126, july 2010. 25

[50] Microsoft Corp. .NET Platform. Available at http://www.microsoft.com/net/. 9

[51] Microsoft Corporation. Dynamic Systems Initiative. http://www.microsoft.-com/business/dsi/. 2

[52] Microsoft Corporation. DCOM Architecture. http://msdn.microsoft.com/-library/en-us/dndcom/html/msdn_dcomarch.asp. 9

[53] Microsoft Corporation. WMI - Windows Management Instrumentation. http://www.microsoft.com/whdc/system/pnppwr/wmi/default.mspx. 13

[54] Microsoft Corporation. CLR - The Common Language Runtime. http://msdn.-microsoft.com/netframework/programming/clr/default.aspx. 13

[55] A. V. Mirgorodskiy and B. P. Miller. Autonomous analysis of interactive systems with self-propelled instrumentation. In *Proceedings of the 12th Multimedia Computing and Networking (MMCN)*, January 2005. 13

[56] Mohammad A. Munawar, Miao Jiang, Allen George, Thomas Reidemeister, and Paul A. S. Ward. Adaptive monitoring with dynamic differential tracing-based diagnosis. In *Proceedings of the 19th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM)*, 2008. 2, 26, 129

[57] Mohammad A. Munawar, Miao Jiang, Thomas Reidemeister, and Paul A. S. Ward. Monitoring multi-tier clustered systems with invariant metric relationships. In *Proceedings of the 3rd Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, 2008. 129

[58] Mohammad A. Munawar, Miao Jiang, Thomas Reidemeister, and Paul A. S. Ward. Filtering metrics for minimal correlation-based self-monitoring. In *IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*, 2009. In press. 129

[59] Mohammad A. Munawar, Miao Jiang, and Paul A.S. Ward. Incremental budget-constrained system modeling and tracking. Technical Report 2009-08, Department of Electrical and Computer Engineering, University of Waterloo, 2009. Presented at HotAC 2009. 129

[60] Mohammad A. Munawar, Kevin Quan, and Paul A.S. Ward. Interaction analysis of heterogeneous monitoring data for autonomic problem determination. In *IEEE International Symposium on Ubisafe Computing*. IEEE Computer Society Press, 2007. 129

[61] Mohammad A. Munawar and Paul A. S. Ward. Better performance or better manageability? In *DEAS '05: Proceedings of the 2005 workshop on Design and evolution of autonomic application software*, pages 1–4, 2005. 129

[62] Mohammad A. Munawar and Paul A. S. Ward. Leveraging many simple statistical models to adaptively monitor software systems. In *ISPA*, volume 4742, pages 457–470, August 2007. 3, 21, 27, 45, 89, 129

[63] Mohammad A. Munawar and Paul A.S. Ward. Adaptive monitoring in enterprise software systems. In *Proceedings of the 1st Workshop on Tackling Computer Systems Problems with Machine Learning Techniques (SysML)*, June 2006. 3, 21, 89, 129

[64] Mohammad A. Munawar and Paul A.S. Ward. A comparative study of pairwise regression techniques for problem determination. In *Proceedings of the International Conference on Computer Science and Software Engineering (CASCON)*, pages 152–166, 2007. 3, 21, 22, 23, 24, 27, 50, 89, 129

[65] Mohammad Ahmad Munawar. *Adaptive Monitoring of Complex Software Systems using Management Metrics*. PhD thesis, University of Waterloo, 2009. 6, 73, 101

[66] Object Management Group Inc. CORBA. http://www.corba.org/. 9

[67] Soila Pertet and Priya Narasimhan. Causes of failure in web applications. Technical Report CMU-PDL-05-109, Carnegie Mellon University Parallel Data Lab, December 2005. 1, 80

[68] Gwilym Pryce. *Inference and Statistics in SPSS: A Course for Business and Social Science*. GeeBeeJey Publishing, 2005. 48

[69] Rice University/INRIA. RUBiS - Rice University Bidding System. http://rubis.objectweb.org/. 75

[70] Felix Salfner, Maren Lenk, and Miroslaw Malek. A survey of online failure prediction methods. *ACM Comput. Surv.*, 42:10:1–10:42, March 2010. 31

[71] Claude E. Shannon. A mathematical theory of communication. *Key Papers in the Development of Information Theory*, 1948. 17

[72] C. Soules, J. Appavoo, K. Hui, D. Silva, G. Ganger, O. Krieger, M. Stumm, R. Wisniewski, M. Auslander, M. Ostrowski, B. Rosenburg, and J. Xenidis. System support for online reconfiguration. In *Proceedins of USENIX Annual Technical Conference*, June 2003. 13

[73] Alexander Strehl and Joydeep Ghosh. Cluster ensembles – a knowledge reuse framework for combining multiple partitions. *Journal on Machine Learning Research (JMLR)*, 3:583–617, December 2002. 18

[74] Sun Microsystems Inc. J2EE Management Specification. http://java.sun.-com/j2ee/tools/management/. 9

[75] "Sun Microsystems Inc.". The Java Virtual Machine Specification. http://java.sun.com/docs/books/vmspec/. 13

[76] Sun Microsystems Inc. The JVM Tool Interface. http://java.sun.com/j2se/-1.5.0/docs/guide/jvmti/. 13

[77] Sun Microsystems Inc. Platform Monitoring and Management Using JMX. http://java.sun.com/j2se/1.5.0/docs/guide/management/agent.html. 13

[78] Sun Microsystems Inc. HPROF: A Heap/CPU Profiling Tool in J2SE 5.0. http://java.sun.com/developer/technicalArticles/Programming/-HPROF.html. 13

[79] Sun Microsystems Inc. Java 2 platform enterprise edition, v 1.4 API specification. http://java.sun.com/j2ee/1.4/docs/api/. 9

[80] Sun Microsystems Inc. JMX — Java Management Extensions. Available at http://java.sun.com/products/JavaManagement/. 9, 14

[81] Ariel Tamches and Barton P. Miller. Fine-grained dynamic instrumentation of commodity operating system kernels. In *OSDI*, 1999. 13

[82] Yongning Tang and E. Al-Shaer. Sharing end-user negative symptoms for improving overlay network dependability. In *Dependable Systems Networks, 2009. DSN '09. IEEE/IFIP International Conference on*, pages 275 –284, 29 2009-july 2 2009. 26

[83] Brad Topal, David Ogle, Donna Pierson, Jim Thoensen, , John Sweitzer, Marie Chow, Mary Ann Hoffmann, Pamela Durham, Ric Telford, Sulabha Sheth, and Thomas Studwell. Autonomic problem determination: A first step toward self-healing computing systems. Technical report, IBM, 2003. 1, 80

[84] Transaction Processing Performance Council. TPC-W – a transactional web e-Commerce benchmark. http://www.tpc.org/tpcw/. 75

[85] Bhuvan Urgaonkar, Giovanni Pacifici, Prashant Shenoy, Mike Spreitzer, and Asser Tantawi. An analytical model for multi-tier internet services and its applications. In *ACM SIGMETRICS*, pages 291–302, New York, NY, USA, 2005. ACM Press. 2

[86] Ian H. Witten and Eibe Frank. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, San Francisco, 2nd edition, 2005. 25, 42, 78

[87] S. A. Yemini, S. Kliger, E. Mozes, Y. Yemini, and D. Ohsie. High speed and robust event correlation. *IEEE Communications Magazine*, 34(5):82–90, May 1996. 2, 25

[88] Hui Zhang, Haifeng Chen, Guofei Jiang, Xiaoqiao Meng, and Kenji Yoshihira. Fast statistical relationship discovery in massive monitoring data. In *IEEE ANM*, 2008. 23