# Adaptive Monitoring of Complex Software Systems using Management Metrics

by

Mohammad Ahmad Munawar

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, 2009

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 supporting networked, transaction-oriented services are large and complex; they comprise a multitude of inter-dependent layers and components, and they implement many dynamic optimization mechanisms. In addition, these systems are subject to workload that is hard to predict. These factors make monitoring these systems as well as performing problem determination challenging and costly. In this thesis we tackle these challenges with the goal of lowering the cost and improving the effectiveness of monitoring and problem determination by reducing the dependence on human operators. Specifically, this thesis presents and demonstrates the effectiveness of an efficient, automated monitoring approach which enables detection of errors and failures, and which assists in localizing faults.

Software systems expose various types of monitoring data; this thesis focuses on the use of management metrics to monitor a system's health. We devise a system modeling approach which entails modeling stable, statistical correlations among management metrics; these correlations characterize a system's normal behaviour This approach allows a system model to be built automatically and efficiently using the monitoring data alone.

In order to control the monitoring overhead, and yet allow a system's health to be assessed reliably, we design an adaptive monitoring approach. This adaptive capability builds on the flexible nature of our system modeling approach, which allows the set of monitored metrics to be altered at runtime. We develop methods to automatically select management metrics to collect at the minimal monitoring level, without any domain knowledge. In addition, we devise an automated fault localization approach, which leverages the ability of the monitoring system to analyze individual metrics.

Using a realistic, multi-tier software system, including different applications based on Java Enterprise Edition and industrial-strength products, we evaluate our system modeling approach. We show that stable metric correlations exist in complex software systems and that many of these correlations can be modeled using simple, efficient techniques. We investigate the effect of the collection of management metrics on system performance. We show that the monitoring overhead can be high and thus needs to be controlled. We employ fault injection experiments to evaluate the effectiveness of our adaptive monitoring and fault localization approach. We demonstrate that our approach is cost-effective, has high fault coverage and, in the majority of the cases studied, provides pertinent diagnosis information.

The main contribution of this work is to show how to monitor complex software systems and determine problems in them automatically and efficiently. Our solution approach has wide applicability and the techniques we use are simple and yet effective. Our work suggests that the cost of monitoring software systems is not necessarily a function of their complexity, providing hope that the health of increasingly large and complex systems can be tracked with a limited amount of human resources and without sacrificing much system performance.

# Acknowledgements

*In the name of Allah, the Gracious, the Merciful.* I am most grateful to God for giving me the opportunity to pursue advanced studies in spite of my little abilities.

I would like to express my sincere gratitude to my academic advisor, 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. Ajit Singh, Dr. Marin Litoiu, Dr. James P. Black, and Dr. Priya Narasimhan for their effort in evaluating this work and for their recommendations for improving it.

I am thankful to Miao Jiang and Thomas Reidemeister for their 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.

I would like to acknowledge the IBM Centre of Advanced Studies, Toronto, for supporting my research financially through a four-year PhD fellowship and for making it possible to validate my work using industry-leading products.

I would like to express my appreciation to my family, especially my wife and parents (both in Pakistan and in Mauritius) for constantly encouraging me, for praying for my success, and for their patience. 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

# List of Tables

# List of Figures

# Chapter 1

# Introduction and Motivation

Computer-based services play a critical role in our society. Many essential tasks in our daily life require use of online services offered by governments, businesses, and other organizations. Examples include e-mail, banking, e-commence, public e-services, *etc.* Likewise, organizations depend on their computer systems to support operations and provide services to users and other organizations. Today, many businesses only offer online services, making them completely dependent on their computer systems. As more-elaborate and more-accessible services become available, the reliance on computer-based services continues to grow. The effect is that the size and complexity of the computer systems needed to support these services, and in particular the software, is increasing.

Software systems are complex because they comprise many inter-dependent components and layers, they implement many dynamic optimization mechanisms, and they are subject to workload that is hard to predict. While the size and complexity of these systems are hidden from the end users, they are visible to those who operate them. The system operators have to ensure that the end users are satisfied irrespective of how large or complex the systems are.

Software systems are especially critical for business entities. In this context, software systems are typically large, complex, distributed, and subject to stringent reliability requirements. They are required to be highly available, operate correctly around the clock, and offer the best level of performance. However, because software systems are not perfect and fault-protection mechanisms are not always present, failures occur. The major manifestations of failure are unavailable systems, exceptions and access violations, incorrect answers, data loss and corruption, and poor performance [117]. The cost of failure is generally high, as it can cause loss of revenues, damage goodwill, even incur penalties for failing to meet service-level agreements. Therefore, businesses spend a significant portion of their information technology budget on managing their computing infrastructure.

Organizations have traditionally relied on human operators to oversee their computing infrastructure, identifying problems, diagnosing their causes, and restoring the system to the desired state. This heavy reliance on human operators is problematic in several ways.

1. It is expensive. System operators that have the knowledge and abilities to cope with large and complex software systems are in short supply and thus expensive to hire. Furthermore, for this solution to continue to work, we need an increasing number of system operators that are more knowledgeable and better-skilled. This is an expensive solution. Statistics show that in 2004 in America, the number of such system operators was approximately 900,000, with the number expected to grow by more than 30% by 2014 [49]. This implies that the annual human-resource cost in America alone is roughly $100 billion and growing.

2. It does not scale. Increasing the number of operators does not make systems management tasks easier. On the contrary, with more people, proper coordination and communication become more difficult, especially in the presence of individuals with varying abilities and degrees of knowledge.

3. It is ineffective. While this solution may provide short-term relief, it fails to address the long-term concern that the complexity of software systems is reaching a level that eludes many human operators [76]. Managing these systems becomes more challenging, as it becomes more difficult to grasp how they work and what the effects of an operator's actions are. Even now, the effectiveness of human operators is questionable: in a recent research study [139], it was found that 40% of system failures are attributable to operator errors.

4. It is not efficient. An operator-driven approach to detecting and resolving errors and failures can be slow. Because of limited resources, it is generally not possible to have human operators maintain permanent, detailed oversight of a system. In addition, manual oversight is time-consuming because of the need to find relevant information in a potentially large amount of complex monitoring data. As a result, it is not unusual for errors and failures to go unnoticed for long periods of time [24], often coming to light only through frustrated users. The end result is reduced system availability, which in turn leads to undesirable business consequences.

It is therefore critical that automated approaches to monitor and manage these systems be developed. This will allow the cost of system management to be reduced, as fewer human operators will be needed, and it will allow larger systems to be built with the assurance that those systems can be managed effectively.

To address this challenge, the idea of self-managing systems has received much attention both from the research community and the industry [18, 40, 54, 94]. The term *autonomic computing* [76] has been coined to refer to self-managed systems. The end goal is to make software systems manage themselves, eliminating or reducing the need for human involvement. System management spans a wide range of activities related to system operation including configuration (*e.g.*, keeping software and hardware inventory and component dependencies up-to-date), performance (*e.g.*, ensuring performance targets are met), security (*e.g.*, access control), accounting (*e.g.*, billing), and problem determination. While the idea of self-management can be applied to all these activities, our work focuses on monitoring and problem determination.

## 1.1 Problem Overview

System monitoring is essential to ensuring proper operation and adequate performance. Effective monitoring allows errors and failures to be promptly detected and their causes identified. We face several challenges in trying to replace human operators by an automated monitoring system. These challenges include:

- Software systems comprise many components and layers with complex interactions between them. Furthermore, the components together may display emergent behaviour, which is not necessarily evident from the properties of the individual components.
- Software systems are dynamic. In particular, their software is often adaptive. In addition, many systems, especially those that are accessible via the Internet, are subject an open-ended workload, which is hard to predict accurately.
- The expected behaviour of software systems is often only defined loosely, except when safety and very high costs are at stake. System operators have some intuition as to what represents acceptable behaviour and performance. However, such knowledge can be difficult to obtain, verify, and encode formally.

- Software systems typically comprise off-the-shelf, generic subsystems (*e.g.*, a database management system, an application server, *etc.*) purchased from independent vendors. Knowledge of the internal structure and inner workings of these subsystems is not accessible in most cases. Even for internally-developed software, the required information may not be documented or may be out-of-date.

- In general, collecting monitoring data is not free. The more data we collect, and the more frequently we collect it, the higher the cost. This cost takes the form of system slow down or data-management overhead.

The goal of this work is to enable automated monitoring and problem determination despite these challenges. A key requirement for such an automated monitoring system is that it should be aware of the cost of monitoring and be capable of controlling this cost while maintaining its effectiveness.

Two important costs are associated with monitoring: human resources and system resources. The heavier the reliance on human operators, the costlier is the solution. Likewise, the more system resources (*i.e.*, computation, memory, storage, and bandwidth) are required, the costlier is the monitoring. Given the large size and complexity of present day systems, both these costs can be high. The aim of this thesis to develop a **cost-aware automated monitoring** system, which can reduce these costs while ensuring that system monitoring remains effective.

### 1.1.1 Enabling Automated Monitoring

Traditionally, several aspects of system monitoring have necessitated human involvement, including configuring and adapting what monitoring data is collected, analyzing the collected data, and from the analysis making inferences about the system's health and faults. It is not practical for human operators to continuously track the system's behaviour and performance (*e.g.*, by continuously visualizing and reading summaries of critical aspects of system operation). In practice, system operators put in place triggers to alert them of conditions that require manual oversight. These triggers are typically based on rules of thumb, which are not necessarily effective. In cases where these triggers work, the remaining tasks still require much time to perform.

Our goal is to reduce human involvement in monitoring and problem determination tasks by having an automated system carry them out or assist in them.

In addition to reducing costs, this approach will increase the effectiveness of the monitoring system by avoiding limitations of the manual approach (*e.g.*, wrong judgment) and ensuring that the system's health is tracked on a permanent basis.

In order to detect errors and failures automatically, the monitoring system needs a way to gauge the target system's health. What is needed is a system model, a characterization of the target system, which can be used to predict its behaviour and/or performance. Building a system model should not necessitate undue effort, expert knowledge, or information that is not available or difficult to obtain.

### 1.1.2 Accelerating Problem Determination

Comparing system behaviour and performance with a system model gives the monitoring system the ability to detect errors and failures. When such anomalous conditions occur, there is a need to pinpoint quickly the cause. The manual approach is often time-consuming, as it involves making sense of complex and potentially large amounts of data. In addition to detection, an automated monitoring system should assist in quickly diagnosing faults in the system. Ideally, the ability to determine problems should not depend on information or expertise that is not readily available.

### 1.1.3 Reducing Resource Requirements

Monitoring comes at a cost. Obtaining monitoring data from a system demands extra resources, including computing power, memory, storage space, and bandwidth. These resources fulfill non-functional requirements and thus need to be minimized. An automated monitoring system has to be cost-aware and yet effective. The level of extra resources utilized needs to kept low while ensuring that the system's health can be assessed reliably and causes of problems determined accurately.

Among the various overheads monitoring entails, the performance overhead is the most critical. It arises from the extra computation needed to measure and capture the monitoring data. Because this overhead directly impairs a system's performance, it is crucial to restrict it to a level that is acceptable.

## 1.2 Scope and Assumptions

A rich variety of computer systems exist, ranging from real-time, safety-critical systems to those created for pure entertainment. Though the solution approach developed in this thesis has wide applicability, we focus on software systems that are component-based and that service short-lived work requests or transactions to a large user base. Examples of such systems abound, including online transaction processing (OLTP) systems, systems providing e-mail and messaging services, stock trading systems, *etc.* These systems are large and complex, making them the right target for evaluating the ideas presented in this thesis.

Software systems make various types of monitoring data available, including log files, execution traces, and management metrics. The focus of this work is on numeric management metrics, which are variables that reflect the state, behaviour, and performance of the target system. We use management metrics to monitor a system. Our system model thus needs to be built with metric data. We rely on the analysis of management metrics to perform both error and failure detection as well as diagnosis. Management metrics may not always suffice for completing these tasks successfully. Nevertheless, as we will show in this thesis, for many problems, they provide pertinent information to speed up these tasks.

In this work the monitoring overhead relates to the measurement and collection of management metrics. We assume that the software systems expose interfaces that allow metric collection to be dynamically controlled. We further assume that the system to be monitored operates under a single administrative domain. As such, the managing system has the privileges to retrieve and control the collection of metrics from the target system.

## 1.3 Thesis Contributions

This work tackles the problem of tracking the health of complex software systems and determining the source of problems that arise in these systems. Specifically, this thesis makes the following novel and significant contribution.

- We solve the problem of monitoring a software system using the management metrics it exposes in an automated way, which reduces human involvement, and in an adaptive way, which reduces the impact on system performance.

  - We present a solution to the problem of how to automatically analyze a system's health without knowing its internal structure or inner workings by devising a system model based on an ensemble of stable statistical correlations between the system's metrics. This modeling approach is suitable for adaptive monitoring, requires little or no human input, is capable of capturing the complex dynamics of software systems, and is efficient to implement.

    We perform an in-depth study of our modeling approach and evaluate alternative options for its implementation, showing the advantages of our approach.

  - We address the problem of how to select a subset of the system's metrics to enable adaptive monitoring using only the metric correlation information. In particular, we present a Minimum Spanning Tree-based metric selection algorithm which, when combined with metric correlation models, enables effective monitoring.

- We devise a diagnosis approach to address the problem of localizing faults using the system's metrics with no *a priori* knowledge of system structure, faults, and metric semantics. Our approach leverages the same system model based on metric correlations which is used to track the system health.

- In order to support our claims, we experimentally validate our solution approach using a realistic test-bed implementing a multi-tier information system, multiple benchmarking applications, and a wide range of faults.

  We show that our approach is effective in detecting faults and, in a majority of cases, provides information that would enable system operators to quickly isolate faults. Further, we provide evidence that metric collection can have a significant impact on system performance, and we show that adaptive monitoring can limit this impact.

The existing work in the area of systems monitoring and problem determination is limited in many respects. In many instances, prior work focuses on tracking specific aspects of a system (*e.g.*, system response time). Often, the problem of error and failure detection is addressed separately from that of diagnosis. Diagnosis approaches are devised by making simplistic assumptions about error and failure detection. For example, there is heavy reliance on basic monitors such as performance thresholds; finding appropriate thresholds without excessive slack is not trivial. Many useful approaches rely on monitoring data that is costly to obtain (*e.g.*, traces); such data is not collected continuously in production systems. Other approaches leverage metric data that is collected by default in production systems for detecting errors and performing diagnosis. The choice of default data is partly motivated by the need to keep the overhead low. While this data is generally insufficient for problem determination, it may even be inadequate for error and failure detection. This thesis develops a solution approach that overcomes these shortcomings. Our work does not assume availability of any pre-existing monitors to detect problems; it entails an integrated approach to detecting problems and determining their causes.

Much of the prior work emphasizes the reduction of the communication overhead of monitoring. Although in recent work mechanisms to reduce the measurement and collection overhead have been proposed, little work exists on how to leverage these mechanisms automatically.

In this work we devise a monitoring approach that automatically controls the collection of the monitoring data while detecting errors and failures effectively, and determining the source of problems when necessary. Our approach can be implemented easily and deployed readily to monitor a large class of existing software systems. Our work makes it possible to create automated monitoring solutions that cost much less than operator-centered solutions and that can significantly improve system reliability.

## 1.4 Thesis Organization

This thesis is organized as follows:

- **Chapter 2:** provides the basic information needed to the understand this dissertation. In particular, it contains definitions of terms that are used in the thesis, covers the basics of management metrics, and gives an overview of distributed component-based software systems.
- **Chapter 3:** discusses the prior research in the area of systems monitoring, adaptive monitoring, and diagnosis.
- **Chapter 4:** presents a high-level overview of how we model the system and how we leverage the system model to reduce the cost of monitoring, to detect errors and failures, and to perform diagnosis.

- **Chapter 5:** describes our experimental setup and our evaluation methodology. It contains a detailed description of the test-bed, the applications, the faults, and the data we use in our experiments.

- **Chapter 6:** contains an assessment the impact of metric collection on system performance and motivates the need for adaptive monitoring.

- **Chapter 7:** presents our approach to characterize the system health for the purpose of automated monitoring. It contains a detailed treatment of the modeling approach, its implementation, and its parametrization.

- **Chapter 8:** provides details on how we assess a system's health. It elaborates on our approach to adaptive monitoring. It presents methods for selecting metrics to track on a continuous basis.

- **Chapter 9:** expounds on our diagnosis approach. It discusses how low-level anomaly data can be combined into useful problem determination information.

- **Chapter 10:** discusses the wider applicability of the solution approach presented in this work, it describes some of its limitations, and presents a summary of other works that extend it.

- **Chapter 11:** outlines the lessons learned in this work and points to promising directions for future work.

# Chapter 2

# Background

In this chapter we provide some background information needed to understand this work. In addition to presenting the terminology used, we cover important aspects of management metrics and provide a brief introduction to systems based on the Java Enterprise Edition framework.

## 2.1 Basic Terminology

The terminology used throughout this thesis follows that of Avizienis *et al.* [10]. For completeness, we reproduce the relevant definitions below.

- 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 *behaviour* 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 behaviour.

- The *service* delivered by a system is its behaviour 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 [87].

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 behaviour).
- The *health* of a system is the degree to which its observed behaviour and performance conform with the expected behaviour 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.

- *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) [20] 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 [130] 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.

## 2.3 Metric-Collection Mechanisms

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 (see related work in Chapter 3).

Management frameworks such as SNMP [20] and JMX [131] 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.

## 2.4 Metric-Collection Overhead

The computation required to update a variable when the underlying measure changes depends on what is being measured. For example, to count how many times an operation occurs, we can instrument Java code as shown in Listing 2.1. At each occurrence of the operation, a counter is incremented. For a remote managing system to collect the counter, additional logic is needed to read, encode, and send the measurement. Similarly, if we need to compute the average time taken by an operation, the instrumentation would resemble that in Listing 2.2; for each operation, two system calls (hence, context-switches) are needed to get the current time. In addition, two variables are needed to record the number of occurrences and the total time taken by a set of operations. If we want to read a metric related to a group of objects (*e.g.*, object pool), the instrumentation would consist of iterating over the objects to compute the measure of interest. Additional overhead arises when mutual exclusion is ensured when updating the variables.

Listing 2.1: Counting occurrences

```
void process() {

        // processing logic ...

        if (metricEnabled)
                operationCounter++;
}
```

Listing 2.2: Measuring average response time

```
void process() {
        if (!metricEnabled){

                // processing logic ...

        } else {

                // get current time
                long start = System.currentTimeMillis();

                // processing logic ...

                // get current time
                long end = System.currentTimeMillis();
                cumulativeTime += end − start;
                operationCounter++;

        }
}
```

Two critical factors determine the overall overhead of metric measurement and collection: first, the quantity being measured (*e.g.*, count, timing, *etc.*) and the number of times the quantity is measured; second, the frequency at which the measured quantities are read and fetched by the managing system. While the first is a function of the amount of work done by the system, the second depends on the managing system.

## 2.5 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) [114], Java Platform Enterprise Edition (Java EE) [136], Distributed Component Object Model (DCOM) [95], and .Net [96]. 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.

### 2.5.1 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

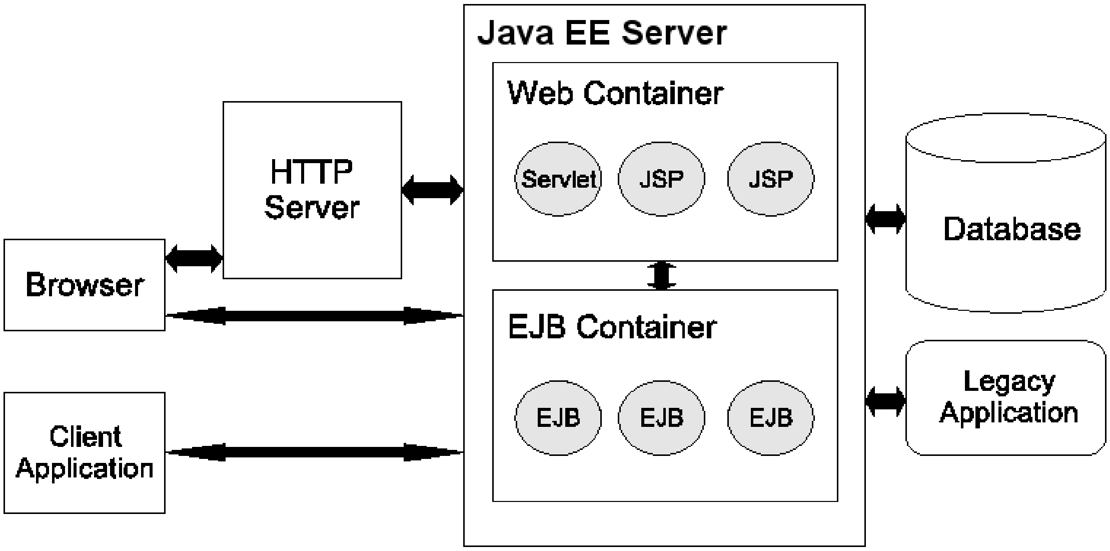


Figure 2.1: Overview of a Java EE-based architecture

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 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.

### 2.5.2 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, behaviour, 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 behaviour, *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 Management Instrumentation (WMI) [97] 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 [19, 101, 138] and dynamic insertion of interceptors between components via hot-swapping [127].

Software systems commonly rely on runtime environments executing above

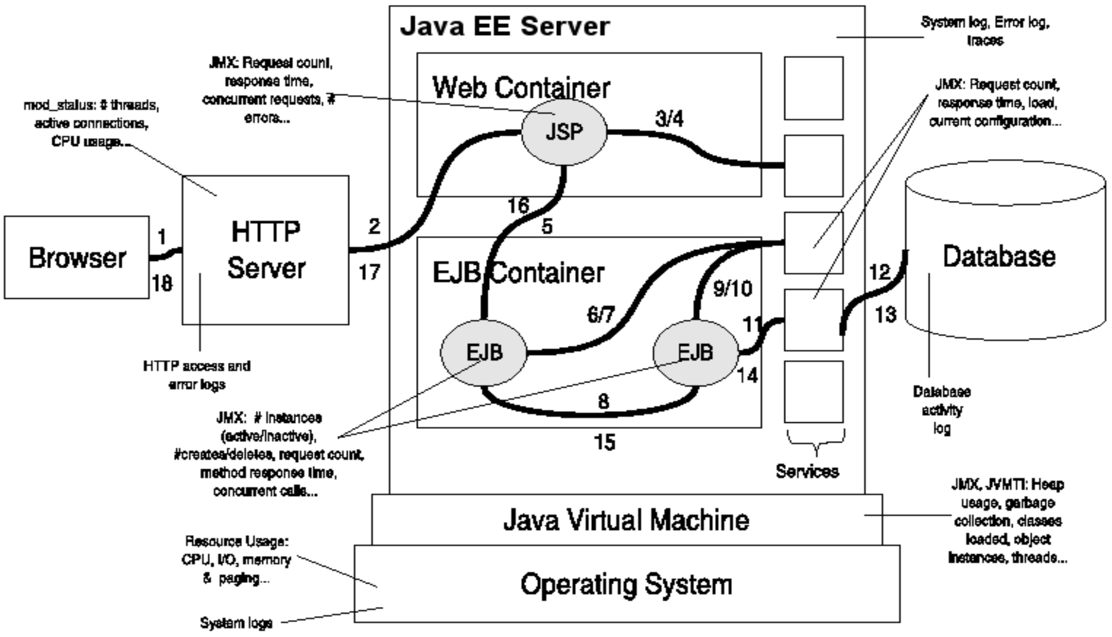

Java EE Server
Web Container
JSP
3/4
EJB Container
EJB
EJB
6/7
9/10
8
11
14
15
16
5
Browser
HTTP Server
1
18
2
17
Database
12
13
Services
Java Virtual Machine
Operating System
mod_status: # threads, active connections, CPU usage...
JMX: Request count, response time, concurrent requests, # errors...
HTTP access and error logs
JMX: # instances (active/inactive), #creates/deletes, request count, method response time, concurrent calls...
Resource Usage: CPU, I/O, memory & paging...
System logs
System log, Error log, traces
JMX: Request count, response time, load, current configuration...
Database activity log
JMX, JVMTI: Heap usage, garbage collection, classes loaded, object instances, threads...

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

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) [132] and Microsoft's Common Language Runtime (CLR) [98]. A Java EE-based system requires a JVM to execute. The JVM provides different interfaces for monitoring. The JVM Tool Interface (JVMTI) [133] 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) [134] 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.*, [135]). 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.*, [64]), 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.*, [55]). 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 [7] of the Apache HTTP server provides a mechanism for collecting such data.

The application server lies at the centre 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 [56]). Most Java EE servers are JMX-enabled [131], 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 [73]).

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, behaviour, 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 application 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.

With this background information, in the next chapter we provide an overview of the prior research on monitoring complex softwate systems and diagnosing problems in them. Much of the prior work has been applied to systems built using component-based frameworks such as Java EE.

# Chapter 3

# Literature Review

A large volume of literature exists on system monitoring, spanning a wide range of application domains, including devices, machines, processes, environmental and social phenomena, *etc.* In this review we concentrate on the monitoring of software systems; in particular, we focus on transaction-oriented software systems that serve large user populations.

The work on software systems monitoring can be organized according to the concerns which monitoring addresses. Two main concerns are ensuring that the target system achieves and maintains a desired level of performance and dependability. The performance of a system is a measure of how well it delivers the correct service. Much work has gone into developing models to track the performance of software systems. We discuss such models later in this chapter.

The dependability of a system is assessed through the attributes of reliability, availability, safety, and security. *Reliability* is the ability of a system to continuously deliver correct service. *Availability* is the ability of a system to deliver correct service when required. *Safety* is the non-occurrence of catastrophic consequences on the user(s) and the environment. *Security* is the ability to avoid improper system alterations, unauthorized disclosure of information, and ensuring the delivery of correct service when needed.

Reliability and availability are important concerns for most long-running software systems. Safety, on the other hand, is most applicable in the context of mission- or life-critical systems. The behaviour and performance of safety-critical systems are generally prescribed by formal specifications, and much effort goes into making sure that the system meets these specifications. In contrast, it is common for software systems for general use, such as distributed information systems, to have more loosely-defined specifications. These systems are often subject to changing, evolving requirements (*e.g.*, because of a competitive business environment). Because of the different nature of safety-critical systems, we do not discuss safety further in this work.

Ensuring security is crucial for most computer systems. Addressing the security challenge is a complex undertaking in its own right, and as such, we view it to be outside the scope of this work. We should, nevertheless, point out that monitoring system behaviour and performance may allow detection of certain forms of security attacks such as denial of service and break-ins accompanied by unauthorized activity. While interesting, this synergy is not explored further.

In order to oversee software systems we require a monitoring infrastructure. We start by providing an overview of prior work in this area.

## 3.1 Monitoring Infrastructure

System management standards typically address monitoring issues such as data representation, communication protocol, programming interfaces, and architecture. For example, standards related to the Simple Network Management Protocol (SNMP) [20] and the Web-based Enterprise Management (WBEM) [37] mandate specific data representation and collection mechanisms. Some standards such as JMX [131] address similar issues in the context of a specific technology (*e.g.*, Java in the case of JMX).

Besides standardization, researchers have developed architectures to facilitate the monitoring of large-scale distributed systems and high-performance computing clusters. Because such systems generate large volumes of data, monitoring needs to be as efficient as possible. For example, NetLogger [44] is a methodology including a set of data collection, analysis, and visualization tools for the end-to-end monitoring of the performance of a distributed system. To reduce network traffic, NetLogger uses a binary event format and provides mechanisms to control the data collection rate. Astrolabe [143] is a general approach to monitoring and managing large-scale systems. Astrolabe organizes nodes into a hierarchy of zones. The amount of monitoring data collected is reduced by aggregating data from the child nodes at the level of the parents. In addition, a peer-to-peer gossip-based protocol is used to share the collected data among the zones. Ganglia [90] is a distributed-monitoring system for clusters and grids that has been adapted for use on wide-area distributed systems. Ganglia is organized as clusters of nodes, and a multicast-based listen-announce protocol is used within each cluster. Representative nodes from the clusters are connected into a tree hierarchy. Aggregation points along the tree periodically poll data from individual clusters and report it to a managing system. Ganglia allows one to specify collection parameters for each metric such as the rate of collection, change-point thresholds, and timeout values.

NetLogger, Astrolabe, and Ganglia focus on reducing communication and storage overheads in a distributed environment. Even though these systems provide visualization tools to facilitate system observation, the onus of reasoning about the target system and configuring the data collected is on the human operators.

Enterprises typically employ centralized monitoring solutions to monitor their computing infrastructure. Examples of such solutions include HP OpenView [52] and IBM Tivoli Monitoring [57]. These solutions provide system operators a control centre from which they can oversee the target system and control what monitoring data gets collected. These solutions also provide facilities to automate monitoring (*e.g.*, setting resource utilization thresholds to trigger alarms). However, configuring and effectively using the tools provided by these solutions is the responsibility of human operators.

Software systems are commonly built using complex software (*e.g.*, database management systems, application servers, and special-purpose applications) from different vendors. These systems are shipped with monitoring tools that can be used independently, without the need for any pre-existing infrastructure. Such tools feature advanced monitoring facilities, since they benefit from the intricate knowledge that vendors have of their products. In spite of their sophistication, leveraging and configuring these tools is still a manual task.

The solutions covered above rely on human operators to analyze the monitoring data and to collect additional data if necessary. This presumes that the operators understand the system and the data exposed. However, because of the system's size and complexity, the operators may fail to understand its functioning and characteristics. Moreover, the amount of available data may overwhelm the operators, thereby reducing their effectiveness and their ability to react in an timely manner.

## 3.2 Basic Approaches to Systems Monitoring

An effective approach to monitoring a software system would be to compare its behaviour and performance against its design-time specification. However, specifications are often loosely defined, incomplete, and not described in terms that can readily be used to monitor the system, making this approach impractical. An alternative approach is to encode expectations of system behaviour and performance. Pip [120] is a monitoring system, that automatically checks whether the expectations spelled out are met during system operation. This approach presumes that the expected behaviour or performance of the system is known. But in complex systems, it can be difficult to know what to expect, even for experts.

In practice, system operators resort to various basic monitors to oversee a system [77]. The most common are low-level monitors such as liveness tests (*e.g.*, pings), periodic log file analysis (*e.g.*, checking for error entries), resource utilization threshold-based alarms, *etc.* Other monitors are system-specific, often tracking aspects that are relevant to the organization. For example, enterprises are interested in the distribution of the types of requests received from their customers that are critical to their businesses. The basic monitors suffer from a number of shortcomings. First, they can be hard to configure. Many of them require domain knowledge and experience with the target system, limiting their general applicability. Second, they have limited detection capabilities not only in terms of the types of errors and failures they can detect but also their severity. Third, they lack the expressive richness to capture the complexity that is characteristic of many software systems.

The ability to monitor a system's health requires a way to determine what the expected behaviour and performance of the system should be under different conditions. It is typical to capture such expectations in the form of a queriable model. We next overview different kinds of approaches for modeling software systems.

## 3.3 Software System Modeling

A model is an abstraction of the system, which allows some aspect of it to be predicted based on relevant observations. A model can be used for monitoring by comparing its output to actual observations and taking unexpected deviations as indication of possible errors and failures.

Prior work in the area of software system modeling can be classified along three axes:

1. What is modeled: system behaviour or performance?

2. In what health state is the system modeled: normal state or anomalous state?

3. What knowledge is assumed about the system: is system structure known or unknown?

### 3.3.1 Modeling Performance

A performance model predicts some attribute related to the performance of a system or its components. Examples of such attributes include resource utilization, throughput, and response time. In addition to their use in capacity planning and provisioning, performance models can help detect performance anomalies.

With knowledge of the system's internals, one can develop analytical models based on first principles. For example, the time taken to execute a program can be computed by adding the time spent in all the functions it invokes. Uysal *et al.* [142] have proposed a simulation-based framework that uses knowledge of the data flow to create such an analytical model to predict performance. Stewart and Chen [129] describe a profiling-based analytical performance model of a multi-component clustered system. Similarly, Shen *et al.* [126] developed and used an analytical model to predict performance of an I/O subsystem. In general, we need to know low-level details such as resource consumption, internal algorithms and associated parameters to build such models. Such information, however, may not be available or may be difficult to obtain. Also, creating empirical analytical models for complex systems is a manual, difficult, and time-consuming task.

Other analytical approaches rely on theoretical principles to model performance. The most common approach for systems shared by multiple users is to apply queuing theory. A queuing model is an abstraction of a system as a set of interconnected queues. A service or a resource is associated with a queue, which holds requests that are waiting to be serviced. Queuing models allow representation of the target system with varying levels of details. For example, one could model a web server using a single queue (*e.g.*, [141]) or model it with elements as detailed as disks and CPUs (*e.g.*, [36, 91]). Queuing models of server software such as HTTP servers are very common (*e.g.*, [34, 91]). Urgaonkar *et al.* [141] have used queuing models to represent multi-tier applications while taking into account user sessions, caching between tiers, and exhaustion of resources. These models are used to tune server configuration parameters such as the maximum number of processes or connections to maximize performance [33] or guarantee a certain level of service [86].

Performance models have a number of limitations. First, they only allow the performance of the modeled part of the system to be monitored. Second, they require manual effort to create. Also, they typically necessitate profiling to estimate model parameters such as service times. However, recent work [128, 149, 151] alleviates this problem by proposing means to estimate these parameters without resorting to profiling. Third, the diagnosis capabilities of such models are limited by the level of details of the target system they capture. In the case of queuing models, the difficulty of solving the models efficiently is a function of the level of details represented.

Some approaches to system modeling do not assume any knowledge of the system's internal structure. We refer to these as *black box* modeling approaches. A black-box model is built by observing a system in operation through its external interface. Here, we are interested in models that are created by using the monitoring data a system provides. Techniques to build such a model pertain to areas of data mining, statistical and machine learning, *etc.* Different types of black box models exist and can be categorized according to what they predict.

Some prior work applies black box approaches to modeling performance. Powers *et al.* [118] study the application of statistical and machine learning techniques, namely auto-regressive and multivariate regression, Naïve Bayes, and tree-augmented Naïve Bayes models, to predict violations of performance targets. Using historical data related to resource utilization, workload characteristics, and performance metrics, they predict future SLO violations or excessive resource utilization. Similarly, Li *et al.* [84] study and compare different time-series modeling techniques for predicting resource exhaustion times in the context of software rejuvenation. The authors use resource and activity data collected from a web server and apply auto-regressive moving average (ARMA) models to predict resource (free memory and swap space) availability. Also, Sahoo *et al.* [123] investigated the use of statistical and machine learning models to predict future failures in machine clusters. In particular, they investigated the use of models such as mean, sliding-window-based mean, autoregressive (AR), moving average (MA), and ARMA.

To apply such black-box modeling, we need to know beforehand what metric is to be predicted. In addition, some domain knowledge is needed to identify the set variables from which predictor variables can be chosen.

### 3.3.2 Modeling Normal Behaviour

Modeling behaviour entails characterizing how a system performs its function. In this work we extend this definition to also include capturing invariant properties of the system while it performs its function. As with modeling performance, there exist two broad categories of modeling approaches: (1) approaches that assume knowledge of a system's internal structure, and (2) approaches that view the system as a black box.

### Structure-Based Approaches

These approaches rely on knowledge of the internal organization and dynamics of a system, including the control or data flow within the system. Such information may be available in a system's design-time documentation, source code, and/or configuration artifacts. When no such information is available or when there is concern regarding how current it is, the system's structure could be inferred from a system's monitoring data. The simplest way to obtain structural information is to trace operations as they execute in the system. Information thus collected may include the components and called operations, timing details, and resources consumed. A standard mechanism to obtain such information in request-oriented systems is the ARM API [73]. Researchers often use custom instrumentation to obtain request traces [23, 24, 25, 83, 78]. Other techniques entail inferring system structure from periodically collected aggregate metrics by applying statistical techniques such as correlation analysis (see, *e.g.*, [5, 11, 17]), containment relationships in response-time metrics (see, *e.g.*, [46]) and time-stamped messages (see, *e.g.*, [5]).

With the availability of structural information, some researchers [13, 24, 78] have used probabilistic finite state machines to describe systems' execution flow. Such a representation allows anomalous execution behaviour to be detected. Barhan *et al.* [13] describe a tool to reconstruct request execution paths along with resource usage information. The authors propose using a probabilistic context-free grammar (PCFG) to represent the execution paths concisely. Symbols of the grammar are components used in servicing user requests. Rules of the grammar correspond to transitions between components, which are assigned probabilities. Barham*et al.* [12] employs clustering to group request paths using a string-edit distance; requests whose path do not fit existing clusters are viewed as potentially anomalous. Chen *et al.* evaluate the use of PCFG for detecting anomalous paths in [24]; anomalous paths are those which fail to be parsed by the learned grammar. Kiciman and Fox [78] also use PCFG and describe an anomaly score based on the learned grammar to detect anomalous behaviour.

Besides execution paths, request traces allow local component interactions to be analyzed. Kiciman and Fox [78] describe the modeling of interactions between system components. Each component's interactions with other component classes (*i.e.*, in-calls and out-calls) are tracked to create a model of normal behaviour. The interactions of each component instance with other components are periodically compared to a reference distribution using the $\chi^2$ test; significant deviations indicate a possible failure. Chen *et al.* [21] also employ the distribution of component interactions to identify anomalous components.

Structure-based behaviour modeling not only allows detection of errors and failures, but it can also aid in fault localization. Nevertheless, it suffers from two main shortcomings: reliance on information that may not be available and/or the requirement of collecting traces, which in general incurs high overhead.

### Black Box Approaches

These approaches to modeling system behaviour do not assume knowledge of the system's internal structure. Hellerstein *et al.* [50] have used time-series models to characterize metrics of interest in a web server (*e.g.*, HTTP request load). This work entails transforming the observed data series into stationary ones by removing trends and cycles (*e.g.*, day-of-week effect). Potential failures are detected by tracking changes in the mean and variance of the processed metrics. The proposed approach is general and can be applied to any metric, provided its behaviour remains in line with the past. In practice, however, systems evolve and the load is at times unpredictable.

Software systems expose many metrics. Not all of these metrics are relevant to every modeling task at hand. Diao *et al.* [35] proposes an architecture and an algorithm to automatically create a quantitative model of a metric of interest. The proposed algorithm discovers relevant metrics by using stepwise regression and outputs a multiple linear regression model. This approach is applicable to cases where metrics of interest are known. Creating models for all available metrics would be computationally expensive.

Bodic *et al.* [14] describes the use of statistical and machine-learning techniques in conjunction with visualization techniques to promptly warn system operators about existing or impending failures. The authors compute page hit counts and page failures periodically, and apply the $\chi^2$ test and Naïve Bayes models to this data to detect deviations from the normal behaviour. They describe an anomaly score for each modeled feature (*e.g.*, page hit count); these scores can be visualized using different colour schemes to direct the attention of system operators.

Chen *et al.* [21] proposes a multi-variate statistical approach whereby the high-dimensional metric data is reduced and tracked via one dimensional statistics. They extract the signal and noise components from the metric data and track the two components using the Hotelling $T^2$ and Squared Prediction Error (SPE) statistics respectively. Anomalies are detected when the statistics deviate significantly from their expected range. The approach proposed in [21] is not robust in the presence of workload variations. To address this shortcoming, Chen *et al.* [22] proposes a fault detection approach which uses Canonical Correlation Analysis (CCA) to subdivide the monitoring variables into two groups: those that vary with the system inputs (*e.g.*, workload) and those that are less correlated with the inputs. The first group is monitored by tracking changes in the correlations; significant deviations signal the possible presence of errors in the system. The second group is assumed to follow a random uniform distribution, and a statistic based on the distribution of normalized values of individual variables is used to track the group. Because the two groups account for all available metrics, this approach allows detection of a wide variety of failures. One issue with the above multi-variate modeling techniques is the lack flexibility; they require a set of metrics to be monitored continuously.

### Metric Correlation Models

The existence of long-term, stable metric correlations in complex information systems and the idea of using these correlations for system monitoring was proposed, during the same time frame, by the author [109] and Jiang *et al.* [65]. This approach is black box in that no information about the system structure or its inner workings is required. Jiang *et al.* propose the use of autoregressive linear regression with exogenous input (ARX) models to capture the metric correlations. They use models with two independent variables and time-lagged versions of two variables. The authors provide an assessment of the error detection capabilities of metric correlations in [66]. ARX models can be hard to interpret, especially when the model coefficients have opposite signs. The authors do not provide any intuition as to why ARX models are preferred over simpler models in all cases. The authors assume the continuous availability of a fixed set of metrics, which are deemed sufficient for problem determination. However, the granularity of the metrics collected by default by system operators limits the extent to which faults can be localized. In Guo *et al.* [45] the authors investigate the use of Gaussian Mixture Models (GMM) to model metric correlations. Though powerful, GMM is expensive, as the model parameters are estimated using the expectation maximization algorithm. The cost of searching for two-variable correlations is $O(n^2)$. In follow-up work [67] Jiang *et al.* discuss two algorithms to speed up the discovery of stable metric correlations at the cost of missing some. The first algorithm groups correlated metrics in clusters, and only searches for stable correlations within each cluster. The second algorithm optimizes model learning by taking advantage of the transitivity of metric correlations; the algorithm approximates model parameters from those that have already been estimated.

Researchers have often used statistical correlations among metrics to understand system behaviour and to locate faults. Brown *et al.* [17] use correlations among metrics to infer dependencies between components of a system. Their approach entails intentionally perturbing the system and the induced statistical correlations are used to analyze potential dependencies. Hauswirth *et al.* [48] leverage correlation information to carry out root-cause analysis. ADMiRe [124] is a tool to analyze system performance. It applies regression analysis to performance data and encodes metric correlation in the form of rules. The tool allows evaluation of expressions involving combination, commonality, and difference between correlation rules from different system configurations. The authors also propose a way to rank regression rules (*e.g.*, new rules that have appeared) to make it easier to spot significant differences. This work is not intended for system modeling; instead, its goal is to enable differential analysis for performance tuning, in particular to identify bottlenecks. Agarwal *et al.* [2] described how correlation among change-points in time-series of different metrics can allow the creation of problem signatures. The presumption in these works is that faults induce correlations, which differs from the view that correlations exist among metrics in a well-behaved system and faults disturb these correlations. We view the two perspectives as complementary.

### 3.3.3 Modeling Anomalous Behaviour or Performance

Rather than modeling the normal behaviour or performance of a system, we can build models to detect anomalous conditions or events. A common example of such conditions is violations of pre-defined performance targets. Sahoo *et al.* [123] propose a rule-based classification approach to predict anomalous events. The rules are inferred by analyzing the most frequently occurring events preceding an event of interest. In a similar effort, Malek *et al.* [88] propose two approaches to predict future failure events with the goal of rejuvenating the system with a complete or a partial reboot. For event data, the authors use a Markov model which takes clusters of events as input and predicts specific failure events. They also employ a non-linear statistical modeling technique named Universal Basis Functions (UBF) to predict failure using time-continuous metrics.

Cohen *et al.* [26] studied the use of machine learning techniques to automatically learn models for performance failure prediction using commonly-collected, low-level system metrics. Their approach entails learning Tree-augmented Naïve Bayes (TAN) models based on small subsets of the available metrics. Learning these models consists of selecting the subset of variables that correlates most with SLO violations and learning the model structure (*i.e.*, dependency relationships between the metrics). The learned models are sensitive to changes in the workload; to remedy this problem, Zhang *et al.* [150] propose extensions whereby, instead of one, a set of models is learned for each type of violation. At any time, the model whose predictions most closely match observations is used. These efforts focus on predicting performance-target violations, which are discrete variables. This approach cannot be readily extended to the prediction of time-continuous metrics because of restrictive assumptions made by basic Bayes models [41].

Other researchers have developed models to detect symptoms of specific faults. Ghanbari and Amza [43] propose the use of Bayesian models to identify specific faults based on results from a mix of models, including metric correlation models. This approach requires that system components and their dependencies be encoded in a Bayesian network. Agarwal *et al.* [3] describe an approach to associate with faults a unique pattern of events, in particular abrupt changes and correlated changes in performance metrics. The work in [26, 150] has been further extended in [27] to define signatures for recurring failures. The presumption is that every unique combination of metrics found to correlate with SLO violations indicates a different type of failure. These signatures can subsequently be used to retrieve past occurrences and associated corrective measures.

## 3.4 Diagnosis

The goal of the diagnosis task is to find the cause of some observed, unexpected phenomenon. In particular, the goal of fault diagnosis is to pinpoint the cause of an observed error or failure. It is not always possible to identify the precise cause automatically because of the limited visibility into and understanding of the target system. However, even in such cases, automated diagnosis can provide pertinent information to help pinpoint the fault location and, in doing so, speed up the manual task of identifying and resolving the underlying cause.

As discussed earlier, one approach to identifying faults is to create signatures leveraging relevant events and metric behaviour that are unique to the faults. This is the approach traditionally taken in the area of network management, where diagnosis has received much attention. For example, Yemini *et al.* [148] propose an approach that requires all managed entities, events they generate, and the dependencies between the entities to be specified in advance. The approach consists of building codes representing events that occur with each known failure. Monitoring entails matching observed events against the pre-identified codes. Such an approach is impractical for modern software systems, because the required information is generally not available. In this section we discuss alternative diagnosis approaches, which do not rely on such information and prior knowledge of faults.

Recent approaches to diagnosis leverage data that is readily available and process this data using statistical and machine learning techniques. For example, models based on execution paths can be used to find components or features that correlate with failed requests using techniques such as clustering [25] and decision trees [23, 78]. It is also possible to diagnose faulty components based on their interaction with other components [21, 78].

Agarwal *et al.* [1] propose an approach that relies on pre-defined application service-level objectives (SLO) to classify the system state as good or bad. Metrics collected during the good state are used to model the expected performance of individual components. When failure occurs, this characterization together with knowledge of the execution flow is used to pinpoint components whose performance deviates from their expected levels.

The Naïve Bayes-based approach of Cohen *et al.* [26] to predict SLO violations, which we described earlier, is particularly useful for diagnosis. It entails correlating low-level system metrics with SLO violations and determining the degree to which each metric contributes to the SLO violations. The identified metrics can help guide system operators to the faulty component.

Diagnosis information can often be inferred from the anomaly detection technique employed. To this effect, we can estimate the contribution of each component found to be anomalous in the total anomaly score computed at the time of detection [14, 78]. A similar approach is to compare a sample of metrics considered anomalous to previous samples that were deemed normal [21, 22]. In this case, diagnosis can be performed by inspecting the degree of change for each metric between normal and anomalous samples. For this approach to work, it is necessary that the samples compared were obtained under the same workload conditions.

Jiang *et al.* [66] propose a technique for fault localization based on metric correlations, whereby components are scored based on the number of perturbed models to which they are associated. However, the authors do not evaluate their diagnosis approach.

In replicated systems it is possible to perform diagnosis by identifying entities whose state or behaviour differs from the norm defined by the majority. Such an approach is discussed in Kiciman and Fox [78] and Pertet *et al.* [116]. Pertet *et al.* combine local, threshold-based anomaly detection with global differential analysis to identify faulty nodes in group communication systems. Kiciman and Fox compare execution paths on different peers to identify application-level faults in multi-tier applications. Similar ideas have been used to carry out offline diagnosis of configuration faults [93, 144].

## 3.5 Reducing the Cost of Monitoring

Monitoring introduces various overheads, which arise from the measurement, collection, handling, and processing of the monitoring data. Existing approaches to systems monitoring differ according to the degree to which they attempt to reduce these overheads. The cost of monitoring is a function of the type of data collected and the collection rate.

It is common for researchers to report the measurement overhead (*i.e.*, the slowdown caused by the extra computation of the monitoring logic)(see, *e.g.*, [39, 19, 101]). However, such information is more difficult to find for industrial software products. In the context of Java EE applications, some figures show that the overhead of monitoring can be high (see, *e.g.* [63]). Lahmadi *et al.* [81] studied the impact of management requests (*e.g.*, requests to retrieve monitoring data) on system performance. Their assessment in the context of JMX and Java EE-based systems shows that the rate of collection can impair performance significantly.

There are two broad categories of work that tackle the reduction of the monitoring overhead, including the adverse effect on system performance. First, most monitoring solutions employ mechanisms to make monitoring as efficient as possible. Second, some monitoring solutions entail dynamically adapting what data is measured and collected. We review this work next.

### 3.5.1 Efficient Monitoring Mechanisms

The most common approach taken to reduce storage overhead involves summarization and pruning. Summarizing consists of aggregating raw data at different time horizons (*e.g.*, day, week, month, *etc.*). Pruning entails discarding data that is no longer of interest. For example, a system operator may be interested in keeping the current week's detailed data, but only want aggregates for periods further in the past. Note that pruning can also be applied to summarized data (*e.g.*, the operator may only want to keep aggregated monitoring data for a year). Monitoring systems such as IBM Tivoli Monitoring [57] provide summarization and pruning capabilities. Other tools such as the Round Robin database tool (RRDtool) [115] also provide such capabilities.

We can reduce the communication or network overhead of monitoring by using compact data representation (see, *e.g.*, NetLogger [44]), by resorting to *in situ* aggregation (see, *e.g.*, Astrolable [143]), and by leveraging efficient communication protocols (see, *e.g.*, the gossip-based protocol in Astrolabe [143]).

We can contain the effects of monitoring on system performance by measuring and collecting less data (*e.g.*, fewer metrics) or by employing sampling. Sampling involves observing a subset of events of interest in order to make inference about the overall population. For example, frequently executed procedures can be identified by checking the call stack using periodic timer interrupts; the alternative is to instrument all procedures to obtain the exact call frequencies. Sampling has been used for code profiling with hardware support [6] and without it [8]. This is also the main approach used to limit the overhead of collecting ARM traces [73].

### 3.5.2 Adaptive Monitoring

Adaptation can be implemented at different levels in the monitoring process. Most commonly, we find work on adaptation at the data collection level and at the measurement level.

Adaptation at the data collection level is typically used to reduce the communication overhead. One approach entails adjusting the rate at which data is collected while satisfying given accuracy objectives. This approach has mostly been used for network traffic analysis [42, 51, 119]. Another approach to control the communication overhead is to define conditions under which the monitoring data should be transferred to the managing system. Diaconescu *et al.* [32] describes a proxy-based adaptive framework for monitoring the performance of Java EE applications. In their approach, component proxies collaborate to decide the most relevant component that should report to the managing system when failure occurs. Agarwala *et al.* [4] investigated the notion of quality of service (QoS) in the context of system monitoring. To support QoS, they propose different classes of monitoring channels, each with different level of details, precision, rate of information, *etc.* Consumers can dynamically subscribe to these channels, thus adapting monitoring data they receive. While the above works make it possible to reduce the communication overhead, they mostly ignore the measurement overhead; the monitoring logic continues to execute in the target system, even though not all of the data is fetched.

The goal of controlling the measurement overhead is to reduce the impact of the monitoring logic on system performance. The instrumentation needed to measure and collect metrics is typically added when the system is implemented. However, current technology allows instrumentation to be retrofitted into an existing system, either offline or at runtime. Such instrumentation can be applied to binary machine code [19, 101], intermediate forms (*e.g.*, Java bytecode [39]), and to higher-level languages (*e.g.*, Java Script [79]).

Traditionally, the onus of configuring what is monitored has been on human operators. There is little work on automatically adjusting what information is measured and collected in order to improve monitoring and fault localization. Some approaches rely on knowledge bases, which have been set up *a priori* by domain experts. The Paradyn performance measurement tool [99], developed by Miller *et al.*, features adaptive, dynamic instrumentation to locate bottlenecks in parallel programs. The tool relies on pre-defined hierarchies of hypotheses to uncover the cause of performance issues. More recently, Kiciman and Wang [80] proposed a framework to adaptively monitor AJAX-based web applications, and which takes advantage of the ability to re-deploy the applications quickly. The authors propose the adaptive instrumentation of client applications to diagnose bugs and bottlenecks based on pre-defined policies.

Some adaptation policies do not require human input. A common form of adaptation policy for analyzing program behaviour and performance is to add instrumentation dynamically by following the flow of control [39, 101]. The idea is to instrument by starting with a given function and recursively instrument the callees. These works, however, only deal with collecting data that is potentially relevant. Analyzing the data is left to humans.

Symantec Indepth [137] is a tool that features the ability to adaptively instrument Java EE applications based on a performance budget, given a pre-defined instrumentation policy. The budget, set as a percentage of the application's response time, controls the amount of instrumentation introduced. The tool provides a number of pre-defined instrumentation policies. Examples include instrumenting methods of standard Java EE components, instrumenting methods that take the longest time to complete and their execution paths, instrumenting the longest running, active execution path(s), and not instrumenting methods that are executed very frequently. Such tools focus on identifying performance problems only. While they may make it easier to find the relevant data, the analysis is left for the system operator.

For network monitoring, the metric sampling rate can be adjusted automatically to meet some accuracy objectives (*e.g.*, to keep prediction errors and variance within acceptable levels) [42, 51, 119]. In the context of operating system monitoring, Seltzer and Small [125] describe an approach to adjust the rate of collection based on the variance analysis of data collected periodically at a fine resolution during the previous day. However, such policies require a computable objective function, whose optimization determines an appropriate sampling rate. This requirement limits the applicability of such policies. In transaction-oriented software systems it is generally sufficient to fix the metric collection rate such that the cost is acceptable and the collected data has sufficient resolution to capture dynamics of interest. In this context, instead of the collection rate, the issue of what metrics to collect is more critical.

A general framework for adaptation entails taking account of what is known about the system health and dynamically configuring monitoring to improve the available information. Irina *et al.* [122] discuss such an approach whereby test probes are dynamically selected to diagnose faulty components. The authors employ Bayesian networks to model probabilistic relationships between test outcomes and states of the entities in the system. Test results are used to update the model and select the most informative tests to execute next. A similar approach to probing a system adaptively is described by Natu and Sethi [112]. This approach relies on a simplified notion of system state (*i.e.*, a node is either up or down) and assumes knowledge of the mapping between probe outcomes and system state. It is non-trivial to extend this approach to management metrics, since it requires defining the system state in terms of the behaviour of the system metrics, which are highly dynamic. In addition, it requires a way to determine how the collection of each metric improves the assessment of the system's state.

## 3.6 Prior Work Limitations

In this chapter we provided an overview of prior efforts in the area of software system monitoring and diagnosis. We discussed various limitations of prior work. A summary of the main points is given below:

- Existing monitoring solutions only provide the infrastructure needed for efficient monitoring. Analyzing and configuring the collection of the monitoring data is the responsibility of human operators.

- A number of monitoring architectures target the reduction of the network traffic generated by monitoring. However, large software systems are often completely hosted in data centres having high speed networks. In such systems, tackling performance overhead on the target system is more important.

- Approaches that involve modeling anomalous behaviour or performance require *a priori* knowledge of these anomalies, which limits their general applicability.

- Some approaches entail modeling the target system in the healthy state. However, much of the existing work either assumes the availability of detailed knowledge of a system's internals (*e.g.*, component dependencies), or the availability of data that is costly to obtain (*e.g.*, traces of execution paths).

- Approaches that use knowledge of system structure often require manual effort to create the system model (*e.g.*, analytical models based on low-level resource usage profiles). On the other hand, black box models tend to abstract too much of the underlying system, and thus they are less useful for fault localization.

- Most approaches for metric-based system modeling do not readily lend themselves to adaptive monitoring; they assume that a fixed set of metrics is always collected. After a model is learned, changing the set of input metrics necessitates that the system model be re-learned.

- Most work on adaptive monitoring focuses on mechanisms (*e.g.*, dynamic instrumentation). Adaptation policies proposed so far are created manually by human experts, are very basic (*e.g.*, instrumentation driven by the execution flow), or have limited applicability (*e.g.*, adapting the collection rate based on an accuracy objective function). There is no prior work on adapting the set of metrics that are tracked to monitor a software system.

- Much work on fault diagnosis assumes knowledge of a system's components, their dependencies, and the events they generate. The information required by the proposed approaches is often not available in large software systems. Scaling such approaches to large systems is also impractical.

- Some diagnosis approaches rely on costly monitoring data such as execution paths. Because of its cost, such data is not collected on a permanent basis in production systems.

- Many approaches assume the existence of independent error or failure-detection mechanisms to label the monitoring data. The labeled data is then used to learn a model using statistical and machine learning techniques. Often, these works assume the existence of pre-existing performance SLOs. If SLOs are not mandated by a service-level agreement, setting them correctly is a difficult task.

- Some diagnosis approaches presume that the data collected by default is sufficient to determine the cause of observed anomalies. However, this is often not the case, as only critical indicators are collected continuously to limit the monitoring overhead. As such, diagnosis based on this data can at best only point in the right direction.

In the next chapter, we present a solution approach that overcomes many of these limitations of the prior work. Our solution approach assumes very little about the target system and collects data on a per-need basis.

# Chapter 4

# Solution Overview

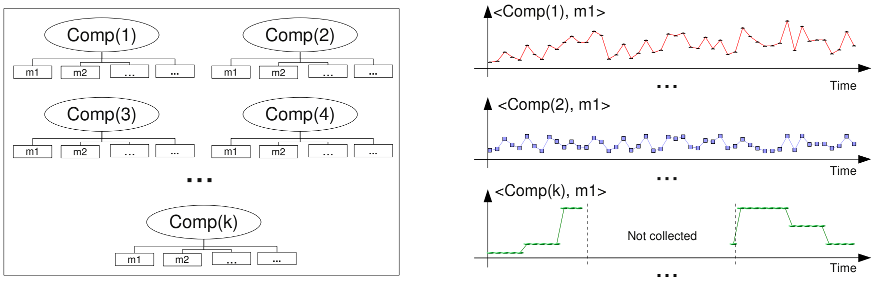
In this chapter we provide a more-precise problem statement and highlight our solution approach. We first present an abstraction of the system to be monitored, which reflects our basic assumptions. Our solution approach applies to any system that fits this abstraction.

## 4.1 System Abstraction

Our basic assumption is that the system comprises components, each of which exposes management metrics. Knowledge of components and their metrics can be obtained from a system's management interfaces, its configuration artifacts, or documentation. This view of the system is depicted in Figure 4.1. Note that with this view, we make no assumption about the system's internal structure.

The visibility of system components through metrics makes it possible to provide a comprehensive monitoring of the system. Moreover, by analyzing the metrics' status, we can assess the degree to which components are anomalous.

We assume that when the collection of a metric is enabled, its values are retrieved at fixed, regular intervals. The repeated recording of a metric's values produces a time-series. Figure 4.1(b) depicts three time series collected at the same intervals. We presume that the collection interval is configured such that significant activity is recorded in each interval and aspects of interest are visible at the corresponding time resolution. For example, in OLTP systems, a collection interval in the order of tens of seconds is sufficient to capture system activity and indicate various trends. In contrast, in a system that services long-running work requests (*e.g.*, scientific applications), a short sampling interval is only useful if there are



(a) (b)

Figure 4.1: System abstraction

metrics that capture the progress of the running programs and the changing state of the system.

In the example depicted in Figure 4.1(b), the collection of metric <comp(k), m1> is disabled for some time, whereas the other two metrics are sampled continuously. We assume that the measurement and collection of the metrics can be controlled (*i.e.*, enabled or disabled) by a managing system. However, if the target system does not provide an interface to enable or disable the measurement of metrics, we can still control the communication overhead by not fetching the values of those metrics that are not of interest.

## 4.2 The Problem

The goal of this thesis is to develop an automated approach to system monitoring that leverages the rich set of management metrics a software system exposes. The monitoring approach should satisfy four key requirements: First, it minimizes human involvement. Second, it is efficient, making it viable for use in a production system. Third, it yields a correct assessment of the system health. In particular, it is effective at detecting errors and failures, and is robust to false alarms. Fourth, it enables fault localization, allowing fast identification of the root cause of errors and failures.

A naïve solution to fulfill these requirements is to collect and analyze all metrics exposed by a software system continuously. This solution, however, is not practical. First, collecting these metrics impairs system performance and imposes other resource demands. Second, this requires means to analyze the collected metrics, without which the collection would be of no avail.

To make monitoring efficient, we adopt a solution approach that entails adapting what is monitored, while ensuring that the ability to determine the health of the system and its components is maintained. In particular, this solution approach consists of changing the set of metrics that are measured and collected dynamically. We discuss the reasons for preferring this solution approach in subsequent chapters.

For adaptive monitoring to work, we need to address the following challenges:

- We need an approach to modeling the system based on management metrics, which:
    - is able to capture complex system dynamics that are reflected in the metrics; covers as many dynamics in the system as possible to provide broad coverage of errors and failures;
    - can be built automatically, without human assistance;
    - can be learned and applied efficiently;
    - enables adaptive monitoring, and can be used with different subsets of the available metrics; and
    - allows the health of individual components to be determined.
- We require an approach to tracking the system's metrics in an adaptive way with the goal of containing the monitoring overhead. Specifically, we need a way to determine what metrics to collect when.
- We need an approach to analyzing the metric data with the goal of localizing faults when errors and failures are detected.
- We need a method to estimate the overhead of measuring and collecting the monitoring data. The cost estimation method has to be practical, requiring little time and effort to apply. It should make use of data that is commonly available. Moreover, the method should be robust to workload variations.

We now outline our solution approach; we provide a detailed treatment of each aspect of the solution approach in subsequent chapters.

## 4.3 Solution Overview

Central to our solution approach is how we model the target system, as it not only determines the extent to which the system's health can be assessed automatically, but it also establishes what level of adaptation of monitoring is possible.

### 4.3.1 Modeling the Target System

In general, there exist two ways to model a system: we can either model the system in the healthy state or model it in the presence of faults. If the non-healthy state is modeled, then the faults need to be known *a priori*, and the monitoring can only check whether those faults have occurred. Learning fault signatures often requires fault injection, which may not always be possible. Because of these limitations, we take the alternative modeling approach of modeling the system in the healthy state.

Modeling the system in the healthy state has several advantages: first, this allows us to ascertain the healthy state of the system at any time by collecting the available metrics; second, data to learn the normal state is readily available; it does not require fault injection and it can be collected from a system in production; third, we can vary the scope of monitoring to ascertain the system's healthy state with different levels of confidence. These advantages make modeling the normal health of the system better suited to adaptive monitoring.

A variety of modeling approaches exists to capture specific characteristics of a system. For example, we can model the behaviour using finite state machines, the state using Petri nets, and the performance using queuing models. These approaches generally require knowledge of the inner workings of the target system. But, such knowledge may not be available; even if it is available, expertise and considerable effort is required to create and parameterize the models. To overcome these challenges, researchers are resorting increasingly to statistical and machine learning techniques to model system characteristics. By leveraging these techniques, we can create models automatically using the available monitoring data, without much involvement of human experts or knowledge of a system's inner workings. This enables larger, more complex systems to be modeled and tracked. We avail ourselves of these benefits by following this modeling approach in our work.

Our goal is to monitor the overall health of a system regardless of specific considerations (*i.e.*, specific states, state transitions, or performance metrics). What we need is to ensure that observations obtained from a system in operation are characteristic of observations we would expect from the system when it is healthy. This allows us to simplify the problem of modeling the system by focusing on dynamics that are indicative of the system's healthy state and that can be captured easily.

A software system has an internal structure, is determined by the logic its software implements and its configuration. Many parts of the system do not change for relatively long periods (*e.g.*, system software). Such systems exhibit stable, long-term correlations among their management metrics. These correlations are invariant; that is, they are not affected by workload variations or the passage of time. Our modeling approach entails creating a signature of the system's normal behaviour by modeling these correlations. We employ regression models to capture the correlations. Software systems expose large numbers of metrics; as such, our global system model is an ensemble of regression models, which represent metric correlations. By combining analyses from the ensemble of correlation models, we can assess the overall health of the system.

In order to check the status of the metrics involved in a correlation, we make predictions with the regression model and compute the residuals with current observations. When the residuals are unexpectedly large, anomalies are detected. While the fact that a correlation continues to hold is not necessarily indicative of the healthy state, its breaking is a sign of disturbance in the system.

Our modeling approach does not allow prediction of specific dynamics in the system (*i.e.*, we cannot explain why specific metrics display a particular behaviour). Creating models to explain the behaviour of individual metrics is generally hard, as it involves understanding and considering the factors that contribute to their behaviour. Instead, our approach involves searching for stable correlations regardless of what induces them. By focusing on correlations, we significantly constrain the complexity that has to be modeled. Our approach only allows us to check whether the modeled metrics are deviating from their normal behaviour.

Besides its simplicity, our modeling approach offers many benefits, including the following:

- Complex system behaviour can be captured using many metric correlations, each of which is characterized easily and efficiently.

- The correlation models are robust; they hold in the presence of varying workload.

- The metric correlations can be identified automatically and modeled by only analyzing the monitoring data. Detailed information about the system's structure, which may not always be available or up-to-date, is thus not required.

- Each model can be used on its own without others. As such, different subsets of the models can be used at different times, making the modeling approach well suited to adaptive monitoring.

- Each individual model allows the status of the associated metrics to be checked. By combining information from different models, it is possible to gauge the overall health of the system.
- As the modeled metrics belong to components, by analyzing results of the ensemble of models, we can localize faulty components.

### 4.3.2 Reducing the Monitoring Overhead

To minimize the monitoring overhead, we have to reduce the amount of metric data that is collected. One approach is to mimic the actions system administrators would take when monitoring a system. Using an approach similar to case-based reasoning, we could determine what data to collect if a fault encountered previously occurred again. Implementing this solution, however, requires a systematic approach to recording and encoding how human operators detect and resolve problems in a system. In addition, this approach only applies to faults that have occurred in the past; unseen faults cannot be handled. In practice, it is only possible to encode a few rules of thumb. Such rules fall short of adequately handling the complexity inherent in the target software systems.

Because of the dependencies in a system, faults tend to affect the behaviour of many metrics. Therefore, to detect anomalous behaviour, it is not necessary to collect and analyze all the metrics exposed by a software system; a small subset of the metrics is often sufficient to notice anomalies. We leverage this insight to devise an adaptive monitoring approach to control the monitoring overhead. Our approach is to adapt monitoring to meet the information needs dictated by our assessment of the system's health. At any point in time, we classify the system health into one of three states: healthy, suspect, or failing. The suspect state is when errors or failures are suspected, while the failing state is when we are confident that errors or failures have indeed occurred. The data we collect is a function of the state in which we believe the system to be. When the system is healthy, we collect a small number of metrics, which together give a reasonable indication of the system's healthy state. Since this small set of metrics may not provide a reliable assessment of the system's health, when anomalies are detected with these metrics, we suspect errors to exist or failures to have occurred. In case of such suspicion, we augment monitoring by collecting and analyzing additional metrics to evaluate the system's health in detail.

Ensuring that a system is in the healthy state constitutes a permanent concern.

We address this concern by tracking a minimal set of metrics continuously. This minimal level of monitoring remains in effect so long as the tracked metrics do not display anomalies. The minimal set is chosen to cover key aspects of the system behaviour and performance. Ideally, metrics in the set should be those that are the most sensitive to faults. System operators may have some intuition as to which metrics meet these criteria. In the absence of such intuition or insight, we can analyze the metric data automatically to select metrics that are most likely to be affected by faults.

When metrics in the minimal monitoring exhibit anomalous behaviour, we suspect errors or failures. At this point, we need to gather additional evidence to validate our suspicion. We thus enlarge the set of monitored metrics to perform a more-extensive analysis of the system's health. This is achieved by analyzing the extent to which the modeled metrics are disturbed. The more widespread the disturbance, the more likely faults exist in the system. If the analysis at the detailed level does not confirm the suspicion, we revert to monitoring at the minimal level to contain costs.

Two factors contribute to the low monitoring cost of our monitoring approach: first, we expect the system to be healthy most of the time, wherein we only incur the low overhead of minimal monitoring; second, although detailed monitoring incurs the high overhead of collecting a potentially large set of metrics, it is only enabled temporarily. The same detailed data is used both for gauging the system's global health and, if needed, for performing fault localization. The high overhead is only incurred for short periods, which when amortized over the duration of the system's execution represents only a small increase in overhead.

We should point out that it is not necessary for the adaptive monitoring approach to be limited to two monitoring levels. Our approach affords much flexibility, in that it makes it possible to devise any adaptation scheme that involves individual metric correlation models.

### 4.3.3 Detecting Errors and Failures

Because we adopt a solution approach with two monitoring levels, detection takes place in two steps. We employ metric correlation models in both steps. To achieve higher sensitivity to faults during minimal monitoring, the output of each correlation model is taken into account. If, for any correlation model, the metric observations do not conform with predicted values, errors are suspected, and adaptation takes place by enabling detailed monitoring.

During detailed monitoring, more metrics are checked using correlation models to confirm or refute the error hypothesis. Each correlation model provides an assessment of the associated metrics. We aggregate results from the available correlation models to gauge the overall health of the system. If corroborative evidence is found in the form of more correlation perturbance, the monitoring system reports the information to human operators.

A regression model may not completely characterize the behaviour of the correlated metrics. In addition, many sources of noise exist in complex systems. Therefore, some of the anomalies reported can be spurious. We allow for this possibility when performing the global analysis by requiring a minimal number of anomalies to be observed before a report for the system operators is generated.

### 4.3.4 Diagnosing Faulty Components

Diagnosis is performed when detailed analysis indicates the likely existence of faults. We leverage our ensemble of metric correlation models to identify faulty components. Correlation models allow tracking of metrics, which belong to components. We can thus assess the health of individual components using the metric correlation models.

Our approach to diagnosis involves assigning anomaly scores to the correlation models and aggregating these scores at the level of metrics or components. The scores are then used to rank and shortlist anomalous metrics or components. The final form of the diagnosis is a list of the top-$k$ anomalous items. System operators can use the list to narrow down the faults quickly.

## 4.4 Monitoring System Overview

Our solution approach leverages metric data to track a system's health and to help localize faults. This approach can be used along with other monitoring solutions that make use of other sources of data such as log files or execution traces. The focus of our work is on leveraging management metrics. Approaches that use log files and execution traces are complementary to our work. They can be useful in cases where external evidence is needed to corroborate the results of the analysis of metrics.

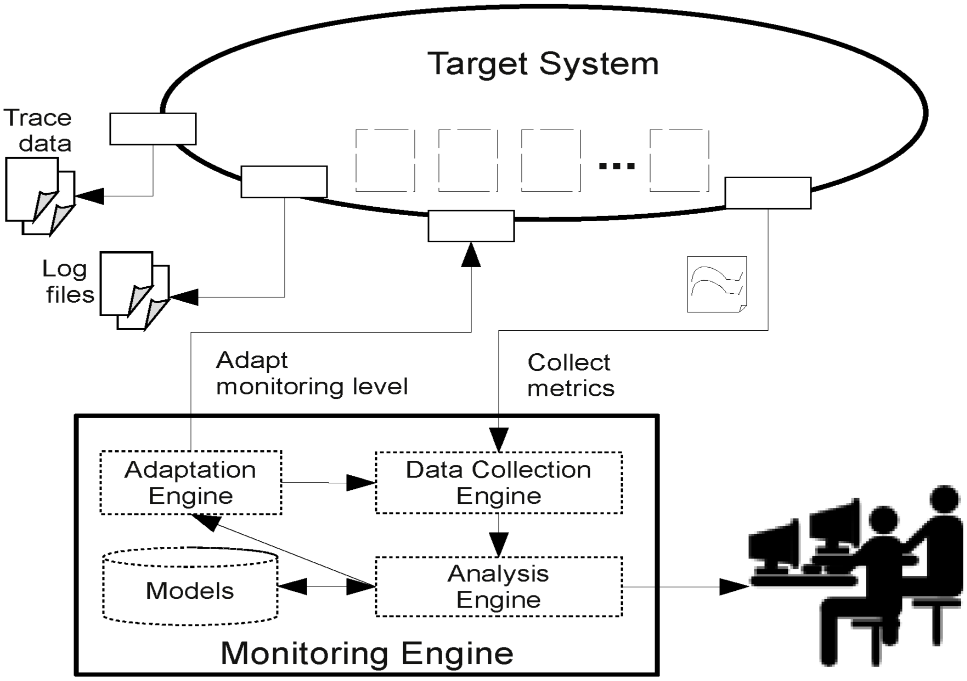


Figure 4.2: System architecture

Figure 4.2 presents a monitoring system which implements our solution approach. The monitoring system collects metric data from the target system to either create a system model or to check newly collected data. Our system model consists of an ensemble of metric correlation models, which are kept in a model store. The correlation models are retrieved from the store when relevant data is available for appraisal. During live monitoring, newly collected data is checked using the applicable models. If anomalies are suspected, an adaptation engine accesses the management interface of the target system to change the scope of monitoring. If adaptation results in more metrics being collected, the applicable models from the model store are used to evaluate the metrics. When sufficient evidence is available to support the error hypothesis, the operators are notified. The notification contains an assessment of the system's overall health together with a diagnosis report. The latter contains details about the components whose behaviour is deemed anomalous. If errors or failures do exist, the system operators can use the diagnosis report to narrow down the fault quickly and take remedial actions.

To evaluate the effectiveness and value proposition of our solution approach, we build a realistic, multi-tier distributed software system based on the Java EE technology. The next chapter describes this evaluation setup and explains how we evaluate our solution approach methodically.

# Chapter 5

# Evaluation Approach

In this chapter 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, adaptive monitoring, and diagnosis.

## 5.1 Evaluation Setup

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.*, [82]) 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.

### 5.1.1 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 [58], which provides the execution engine for Java EE applications. To support long-term data persistence, we make use of the DB2 [59] 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 5.1. All entities are connected via a Gigabit LAN. The setup in Figure 5.1 can be scaled up by adding more application servers or databases, and separate web servers. In related work [103, 104] we have extended this basic setup to include multiple application servers.

### 5.1.2 Applications

We use our target platform to execute several existing applications that mimic functionality implemented in real transaction-oriented software systems. Although these applications vary in size and functionality, and have been developed by different organizations, they share common characteristics. First, they have been built

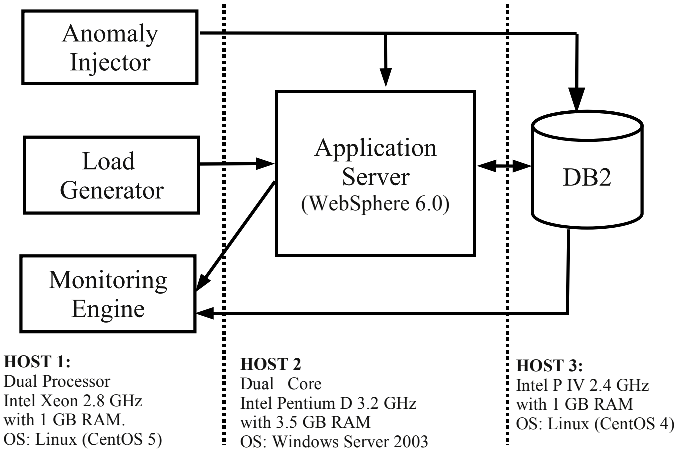


Figure 5.1: Experimental setup

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 [60] 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 [121], 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 [140] is a performance benchmark specification designed to evaluate web-based transaction-oriented systems. We use a servlet-based implementation of the

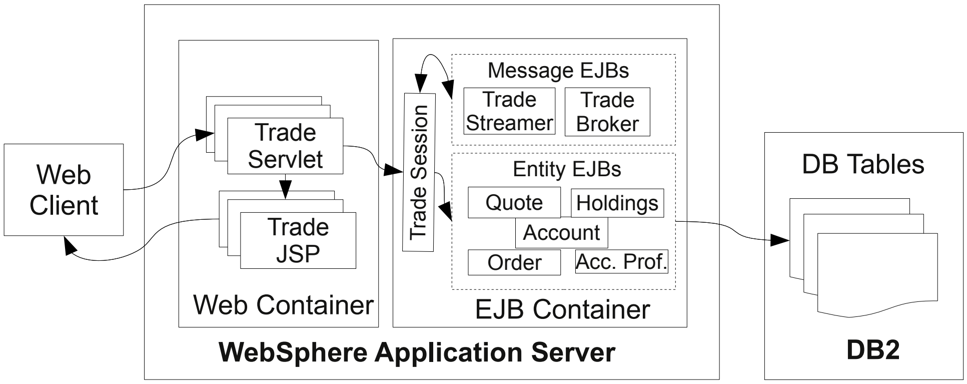


Figure 5.2: Overall structure of the Trade application

specification. The application implements the functionality of an online retail store, allowing users to browse and purchase items. The benchmark can be configured to use workload profiles, which correspond to different proportions of "browse" to "buy" web interactions.

### Trade

Trade [61] 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 5.2. 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.

### 5.1.3 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.

### 5.1.4 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

[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 5.1: Examples of metrics collected

transactions in our applications are short-lived; when the system is not overloaded, 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 [146] 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.*

### 5.1.5 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 5.1 lists a few examples of metrics included in our data sets.

### 5.1.6 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.

## 5.2 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 behaviour, 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.

## 5.3 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.

| **F**ault 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 5.2: 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. In each category, we have several classes of faults. A summary of the faults we use in our experiments is given in Table 5.2 and further details are provided in the following sections.

### 5.3.1 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 $e_{\text{prob}}$ 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 $d_{\text{len}}$ time units by executing extra cpu-intensive logic. To configure these faults, we specify a component, one of its methods, the delay-loop duration $d_{\text{len}}$, and a probability of activation, $d_{\text{prob}}$, 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 $d_{\text{len}}$ 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 $l_{\text{lock}}$ fraction of every $l_{\text{interval}}$ time interval during the fault-injection period.

In our experiments we configure our application faults using the parameters listed in Table 5.3. The tasks of error detection and diagnosis are more diffcult 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 effets similar to that of a probabilistic fault.

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

Table 5.3: Fault parameters

### 5.3.2 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.

In this chapter, we presented our evaluation setup, including a multi-tier Java EE-based test-bed, the monitoring infrastructure, and the data used in our evaluation. We outlined the methodology for evaluating our solution approach. We also described the faults we have developed to check the response of a monitoring system that implements our solution approach. The evaluation test-bed and methodology described in this chapter are used in the following chapters to validate our claims with respect to our solution approach. Prior to dwelling into the details of our solution approach, in the next chapter we use the testbed to show the impact of metric collection on system performance.

# Chapter 6

# Cost of Monitoring

Continuous monitoring is essential in ensuring that a software system operates as desired. We can oversee a system's health by retrieving and analyzing the monitoring data it exposes. Monitoring should allow anomalous conditions or events in the system to be detected quickly. Furthermore, monitoring should allow the causes of anomalies observed to be identified rapidly, making it possible for remedial actions to be taken and the system to be returned to a healthy state in a timely fashion.

Monitoring, however, is not free. Obtaining the monitoring data and analyzing it requires resources, which need to be diverted from their use in performing the system's main function. A software system is primarily intended to serve a specified purpose; a well-monitored system that cannot perform its functions as desired is less useful, if useful at all. It is, therefore, critical to quantify the overhead of monitoring a system and to reason about whether the cost is justified.

Estimating the cost of monitoring is particularly important in systems that are subject to partial failures. Component-based systems that implement multiple services are more likely to suffer from such failures. In such systems, it is essential to have the ability to detect errors and identify faulty components without unduly affecting parts of the system that are functioning correctly. In this context, systems monitoring needs to be cost-sensitive; the cost needs to be minimized while ensuring that the objectives of monitoring are met.

Enabling cost-sensitive monitoring requires that we have the means to estimate the monitoring cost. As described in Chapter 2, monitoring involves various types of overhead. It reduces system efficiency; in particular, the collection of metrics causes additional, non-functional code to execute (*e.g.*, logic for measurement and aggregation, formatting and dispatching, *etc.*). This requires computation and memory resources, which are no longer available for the monitored entity to use. This specific cost can be measured directly by quantifying the effect of monitoring on system performance. In addition, resources are needed to transmit, store, and process the collected data. While the target system may not incur this additional cost directly, it nevertheless adds to the overall cost of the target system because these resources have to be provisioned.

In this thesis we focus on the measurement and collection overhead because of their direct impact on a system's operational efficiency. This overhead can be estimated from readily available performance metrics. While we do not consider the other overheads directly, we should point out that efforts to reduce the measurement and collection overheads often translate into smaller ancillary overheads such as communication, storage, and processing. The other overheads are more difficult to quantify objectively. Communication and storage costs often depend on the way the monitoring data is encoded. Likewise, the processing needed to analyze the data is highly technique-specific. Furthermore, the other overheads may not be as constrained as the computing resource of the monitored entity. For example, in an enterprise context, network bandwidth and storage resources are often over-provisioned; likewise, computing resources needed for the offline analysis of the collected data can be added easily.

## 6.1 Measuring the Performance Overhead

We can estimate the performance overhead of monitoring by estimating the extra computation needed by the measurement and collection of a set of metrics. To this effect, we can either view the target system as a white box or consider it as a black box. A white box view entails an analytical approach, whereby we assume knowledge of the monitoring logic and availability of data to estimate the associated computation overhead. A black box view, on the other hand, entails an empirical approach to estimate the performance cost without knowledge of the system's internals.

### 6.1.1 Analytical Approach

The analytical approach requires that we first identify the various types of metrics. For each type, we need to estimate the computation needed for each update and each collection. The cost of collecting a metric for a particular sampling interval is a function of the number of updates, the computation needed for each update, and the computation required for reading, packaging, and dispatching the metric value. When collecting a group of metrics, a fixed global cost may also be incurred (*e.g.*, a data structure containing metric values may need to be time-stamped or be assigned a global identifier). Further to this detailed cost analysis, we need to derive a figure for the overall overhead, and this figure has to be translated into terms that system operators can understand easily.

The analytical approach is cumbersome in many respects. First, it requires that we be aware of the details of the monitoring logic (*e.g.*, *via* access to the source code). Second, while monitoring the system, we not only need to collect the values for each metric, but also additional data (*e.g.*, number of updates) to allow computation of the overhead for each metric. Not all software systems expose such data. Third, in addition to estimating the various elements of the computational cost of monitoring, we need to express the overall cost in terms of impact on the system performance, which is non-trivial.

### 6.1.2 Empirical Approach

The alternative to the analytical approach is to estimate the computational cost of monitoring empirically. This cost is more-naturally expressed relative to a measure of the system performance. If the performance of the system is measured in terms of response-time, then a percentage increase in response time attributable to monitoring can be used as an estimate of the cost. Such measures are not only understood more easily by system operators but they also make it easier to express requirements of what monitoring cost is acceptable.

One option is to express the performance cost as a percentage of the mean response time in the system. This, however, has an important shortcoming. It is difficult to separate the performance overhead from other factors such as queuing time, especially in the presence of a varying workload. To address this issue, our solution is to abstract the target system as a queuing model and estimate the cost by quantifying the change in the service demand, which is the mean time spent by user requests at a resource.

Queuing models can be built by assuming high level knowledge of the system (*e.g.*, the connections between subsystems such as application server and database). While elaborate queuing models have been proposed in the literature to predict system performance accurately, for our purposes it is only necessary to obtain a good estimate of the relative change in performance with different levels of monitoring; our goal is not accurate performance prediction.

In this work we employ operational analysis [31, 92] to approximate the service demand. Operational analysis defines basic relationships among measurable performance quantities (*e.g.*, length of observation period, number of arrivals, busy period, number of completions) based on a few verifiable assumptions. In making use of operational analysis, we obviate the need to make distributional assumptions on the arrival rate of requests and service time and to solve complex queuing models.

Collecting metrics causes the service demand to increase; the increase is generally proportional to how often, how much, and what data is collected. To estimate the overhead of collecting a set of metrics, we first create a baseline for the service demand using performance data obtained while all monitoring is disabled in the system. We then estimate the service demand using data obtained while a set of metrics are collected periodically. The performance overhead of monitoring is equivalent to the increase in the service demand, which we express as a percentage change.

This empirical approach is simple, relies on readily available performance data, and has the advantage of being applicable in an online setting.

## 6.2 Experiments and Analysis

To illustrate our approach, consider the system supporting the Trade application shown in Figure 5.1. In our testbed, only the WebSphere application server allows fine-grained, metric-level control over the collection of metrics. We thus focus the estimation of the measurement and collection overhead on the application server. In the evaluation below, we change the set of metrics collected in the application server while collecting a fixed set of metrics from the database and the operating system.

To estimate the performance cost of monitoring on the application server, we measure the following quantities in a given observation period $T$:

- $B$, the total time during which the CPU is busy during a period $T$, which can be computed by multiplying the CPU utilization $U$ by $T$.

- $C$, the total number of requests completed during $T$, which is equivalent to knowing the throughput $X = C/T$

Table 6.1: Service demand with different monitoring configurations

| ***Configuration*** | *Mean Service Demand (ms)* | *% Increase from baseline* |
|---|---|---|
| None (baseline) | 3.81 | - |
| Minimal | 3.88 | 1.82 |
| Detailed | 4.28 | 12.09 |

- From the above two quantities, we can compute the mean service demand $D$ using the relationship:

$$D = \frac{B}{C} = \frac{U \times T}{C} = \frac{U}{X}$$

This operational law is known as the *service demand law* [92].

We carry out experiments with different monitoring configurations and estimate the resulting performance overhead. Each experiment comprises a warm-up period to eliminate initial transient effects. We use the `httperf` [102] open-loop workload generator to simulate user activity and enforce exponential arrival time between requests. For each monitoring configuration, we execute experiments at seven request load levels, which correspond to different degrees of system utilization. We repeat each experiment five times at every load level. Results presented here are the mean values obtained over the five repetitions.

Figure 6.1 shows the mean service demand for the CPU resource at the application server at increasing load levels under different monitoring configurations. As noted in Chapter 5, metrics are collected every 10 seconds. Each point on the chart is obtained by averaging five 15-minute runs of the system (excluding warm-up). “None” in the figure refers to disabling monitoring, while “Detailed” denotes a level where all available metrics are collected. Of the metrics collected at the detailed level, roughly 500 are active (*i.e.*, their values are updated as a result of activity in the system). At the “Minimal” level, we collect metrics related to the application's web interface. The 60 or so metrics collected include request counts, the number of concurrent requests, response time, and failure counts of all dynamic web pages.

The web interface is the only way our simulated users access the application functionality; as such, these metrics provide a good indication of the overall health of the system. Table 6.1 presents the average service demand estimated at the different monitoring levels.

The results show that increasing the monitoring level creates significant overhead, which accounts for 12% of the service demand at the detailed level. This

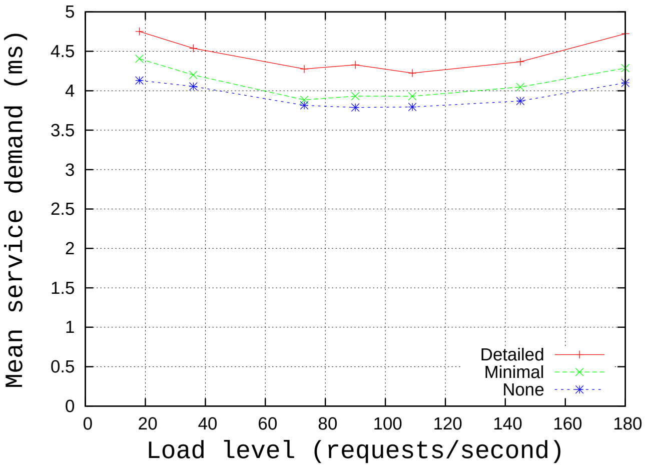


Figure 6.1: Effect of monitoring configurations on mean service demand

overhead will likely not be acceptable on a continuous basis in production systems. In general, in these systems performance is regarded as being more critical than concern for faults that occur rarely. Nevertheless, the high level of overhead may be acceptable if it is incurred for short periods of time. Minimal monitoring, on the other hand, has a small effect on performance because the set of metrics collected is small. Such a minimal level of monitoring can be maintained on a permanent basis without sacrificing much performance. These results bring to light a trade-off between the overhead of monitoring and availability of information about the system's health. If a system's health could be gauged, even approximately, using a small set of metrics, then one approach to controling the monitoring overhead is to rely on this small set so long as the system is deemed to be healthy; additional monitoring can be activated when the system's health requires further investigation. In Chapter 8 we present our adaptive monitoring approach, which builds on this idea.

## 6.3 Summary

In this chapter we show that the cost of monitoring a software system using its management metrics can be high; incurring such cost on a continuous basis may not be acceptable in a production system. We employ an empirical approach to estimate the performance overhead of metric collection based on principles from queuing theory. Using this approach we show that enabling and collecting all metrics exposed by the application server in our test-bed can reduce performance by as much as 12%. This overhead adds to the overheads related to the network usage, storage, and processing needed to make use of the collected data.

Our evaluation serves as motivation for the need for an automated, adaptive monitoring approach to reduce the monitoring overhead while still making it possible to effectively monitor the target system. A pre-requisite for enabling automated, adaptive monitoring is a system model; the next chapter presents our system modeling approach.

# Chapter 7

# System Modeling

A prerequisite for monitoring a system's health is a characterization capturing the system's expected behaviour or performance. For example, a basic characterization of a software system may consist of the maximum utilization levels for key system resources such as CPU, memory, and disk space. We refer to such a characterization as a system model. The system model provides a way to detect unexpected conditions. Therefore, system modeling is the most critical building block needed for our automated adaptive monitoring approach.

Creating a system model is a difficult task because not only do systems that we target have complex structure and behaviour, they also are subject to external stimuli that are dynamic and difficult to predict. In this work we do not assume availability of information about the target system's structure and internal dynamics, nor do we assume *a priori* knowledge of faults the system can experience. The only information at our disposal is the set of system components, their metrics, and the metric values. Despite these constraints, our system modeling approach needs to meet the requirements spelled out in Chapter 4.

A system typically exposes a large number of metrics. If we knew in advance which aspects of a system needed to be overseen, we could select and model specific metrics that capture those aspects. With some domain knowledge, we may be able to identify some key aspects that require oversight. Nevertheless, we do not assume such knowledge. Our goal is to track the general health of the system, without emphasis on specific aspects. Thus, the more metrics we can track with our system model, the more comprehensive the monitoring will be.

One approach to model the system is to use a monolithic model, which takes all available metrics as input. For example, we could employ multi-variate statistical techniques. Such an approach, however, lacks flexibility; it requires tracking to be performed using the same set of metrics which were used to create the model. As such, this approach does not readily lend itself to adaptive monitoring.

Our solution approach is to use an ensemble of metric models, each of which provides an assessment of a small number of metrics. By analyzing all such metric models, we can determine the health of the system and also perform fine-grained analysis to localize faulty components.

The basic approach to modeling the behaviour of a metric is to define an acceptable range for its values. Figure 7.1 illustrates an example of a threshold that delineates the maximum value a metric can take. Thresholds are simple to grasp, and system operators often have some intuition as to what values constitute reasonable limits for certain metrics (*e.g.*, those related to key system resources). However, these represent a very small portion of the overall set of available metrics. In general, finding appropriate thresholds is a difficult task.

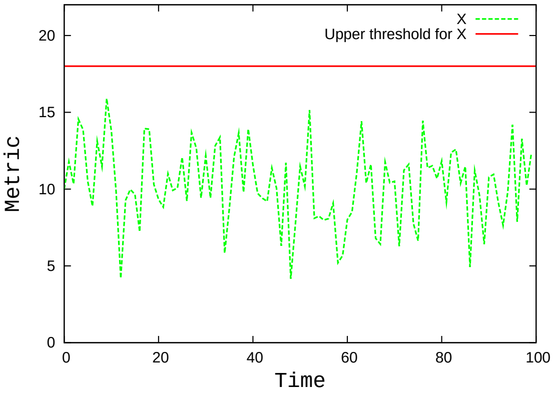


Figure 7.1: Using simple thresholds to track metrics

Metrics of complex software systems are mostly non-linear, reflecting the dynamic nature of the workload as well as the system's internal complexity. Simple mechanisms such as static thresholds fail to capture this non-linearity. Thresholds can only delineate an operating range within which the metrics are expected to vary. If the range has too much slack, it will fail to detect some errors or failures. On the other hand, a range that is too tight will cause many false alarms. The consequences are reduced effectiveness or lower confidence in the monitoring system. Therefore, we need more-general means to track metric behaviour.

## 7.1 Using an Ensemble of Metric Correlation Models

It is difficult to predict a metric's volatile behaviour based on its past values alone, as many factors potentially generate that behaviour. However, a system's structure, as determined by its software (*i.e.*, its code), configuration, and internal dependencies, constrains the system's behaviour; the resulting regularity is reflected in the metrics. Furthermore, when monitoring a system, we are not always interested in the actual behaviour of the metrics (*i.e.*, the different values they assume), especially when we do not have any intuition as to what the metric behaviour should be. In such cases, it is sufficient to determine whether the metric's behaviour is what we would expect if the system was in the healthy state.

On the basis of the above observations, one simple but powerful approach to characterizing a system's normal behaviour is to identify stable correlations among the metrics it exposes. These correlations should be invariant to the workload and the passage of time. Figure 7.2 shows the behaviour of two metrics whose behaviours follow each other closely; one of the metrics is the same as the one shown earlier in Figure 7.1. We observe that one metric can serve as a reference for the other, which allows us to track the metrics' behaviour more accurately than using a threshold.

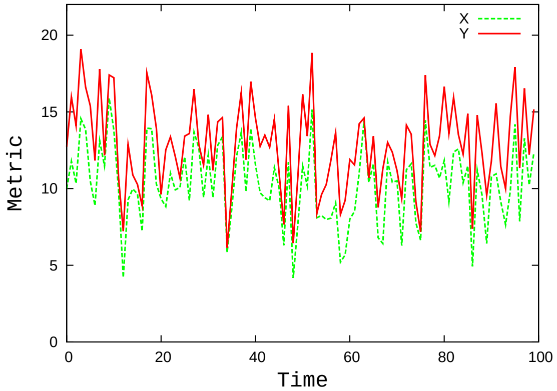


Figure 7.2: Using correlations to track metrics

Causal metric correlations are induced by the many dependencies in the system (*e.g.*, correlation between calls to functions `f()` and `f_sub()`, where the first function always executes the second). Correlations can also be incidental, in which case the correlated metrics vary together because their behaviour is induced by a common factor. Irrespective of how co-variations arise, metric correlations are easier to model than the behaviour of the individual metrics. To appreciate the benefits of this modeling approach, consider the system shown in Figure 7.3. For simplicity we assume that there is a single component in the system. The figure shows that the workload metric, which captures the work submitted to the system, displays behaviour that would be hard to model on its own. Moreover, we see that Metric 2 of the component displays some other behaviour. The component's internal logic and optimization transform the workload behaviour into a more complex behaviour, which would be hard to predict on its own. Despite the complexity in the individual metric behaviour, we can readily model the correlation between the workload metric and Metric 1. The aspect represented by Metric 1 is not affected by the component's behaviour; as a result, Metric 1 is just a reflection of the workload metric. Similarly, the correlation between Metric 2 and Metric 3 is modeled easily; both these metrics are subject to the same complex component behaviour. Therefore, we can make some judgment of the metrics in this system without having to describe the evolution of the individual metrics. Actual systems are much more complex than what is shown in Figure 7.3. The richness of the logic and optimization implemented by the multitude of system components and their inter-dependencies can give rise to very complex metric behaviour. Our approach avoids the need to capture this complexity to monitor the system, albeit at the cost of restricting what we can do with the learned models.

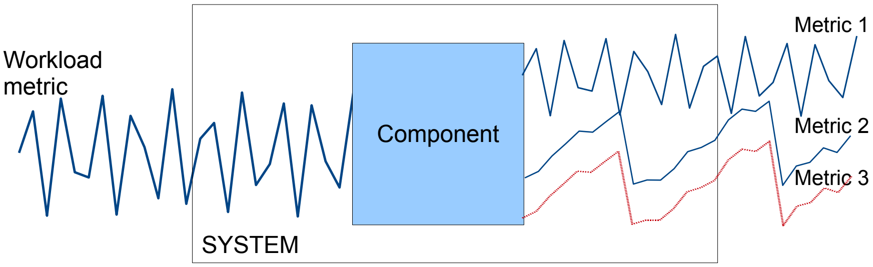


Figure 7.3: Capturing complexity through metric correlations

Our modeling approach is blind to the semantics of the metrics and the relationships among them. We capture stable, long-term correlations that are representative of the system's healthy state. We model them mathematically to enable prediction and anomaly detection. The set of correlation models we identify forms an ensemble, which constitutes our model of the system in the healthy state. The premise of our monitoring approach is that the metric correlations hold during normal operation and that some of them will fail when faults occur. By identifying mismatches between metric observations and predictions of the correlation models, we can detect errors and failures, and we can also identify faulty components.

A correlation model can relate two or more metrics. However, identifying metric correlations becomes exponentially costly in the number of metrics involved; the cost of identifying $k$-metric correlations from a set of $n$ metrics is $\mathrm{O}(n^k)$. In this thesis we only consider correlations between two metrics. Besides reducing the model identification cost, correlations between two metrics are easier to interpret.

Any modeling technique that can capture correlations among metrics well can be used with our approach. The requirement is that the generated models fit the data well, allowing maximum sensitivity to anomalies and minimum vulnerability to false positives. A model that lacks fit may not only produce false positives but also fail to detect anomalies; likewise, a model that overfits the data may detect anomalies in the presence of normal variations. In addition, our choice of modeling technique is restricted by the need to keep the computation and memory cost of modeling low. This is important because we can have a large number of correlation models.

We employ regression models to represent metric correlations because of their simplicity and efficiency. In its simplest form, a regression model is a mathematical model which predicts one dependent variable using one independent variable. More elaborate models may include more than one independent or dependent variable. Many forms of regression can be learned efficiently. For example, closed-form solutions exist for solving linear regression using the method of Ordinary Least Squares. Also, regression models have a very compact form, only requiring information about the variables and their coefficients.

## 7.2 Identifying Stable Metric Correlations

Figure 7.4 presents a high-level depiction of our approach to creating a system model. Identifying the correlation models requires that we obtain time series of the system metrics while the system is in the healthy state. The collected data should cover a period of time which is sufficient to capture representative behaviour of the system. We use this data to identify stable, long-term metric correlations.

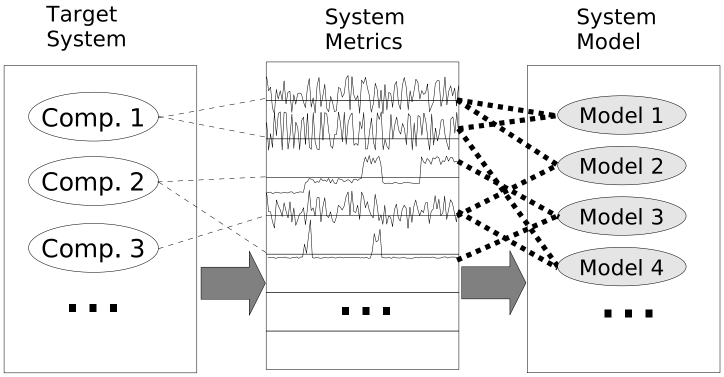


Figure 7.4: Approach to system modeling

The process of identifying stable correlations and using them to monitor the system is summarized in Figure 7.5. The main steps involved in identifying stable metric correlations are to cross-correlate the metrics, find models that fit the data well, and validate the models by applying them to data collected from the system in the healthy state. For online tracking, we apply the learned models to samples of the associated metrics and check for unusual observations. The consistent detection of such unusual observations implies that a correlation no longer holds (*i.e.*, the correlation is perturbed). We discuss details of the model identification and tracking process in the following sections. We elaborate on the global assessment of the models for error detection in the next chapter.

Identifying metric correlations require considering $n(n-1)/2$ (*i.e.*, $O(n^2)$) pairs of metrics for modeling, where $n$ is the number of metrics. While this cost appears to be high, two key factors make our modeling approach practical. First, we employ modeling techniques that are computationally efficient. Second, model identification is fully parallelizable; each correlation model can be learned independently of the others, provided the metric data can be shared. As discussed in Chapter 3, the modeling cost can be further reduced by resorting to approximation algorithms proposed in Jiang *et al.* [67].

In this work we employ regression modeling techniques which are computationally efficient. Specifically, the basic technique we use is simple linear regression, which is the most efficient regression technique. We also investigate more-powerful variants of linear regression, which can capture non-linear metric behaviour.

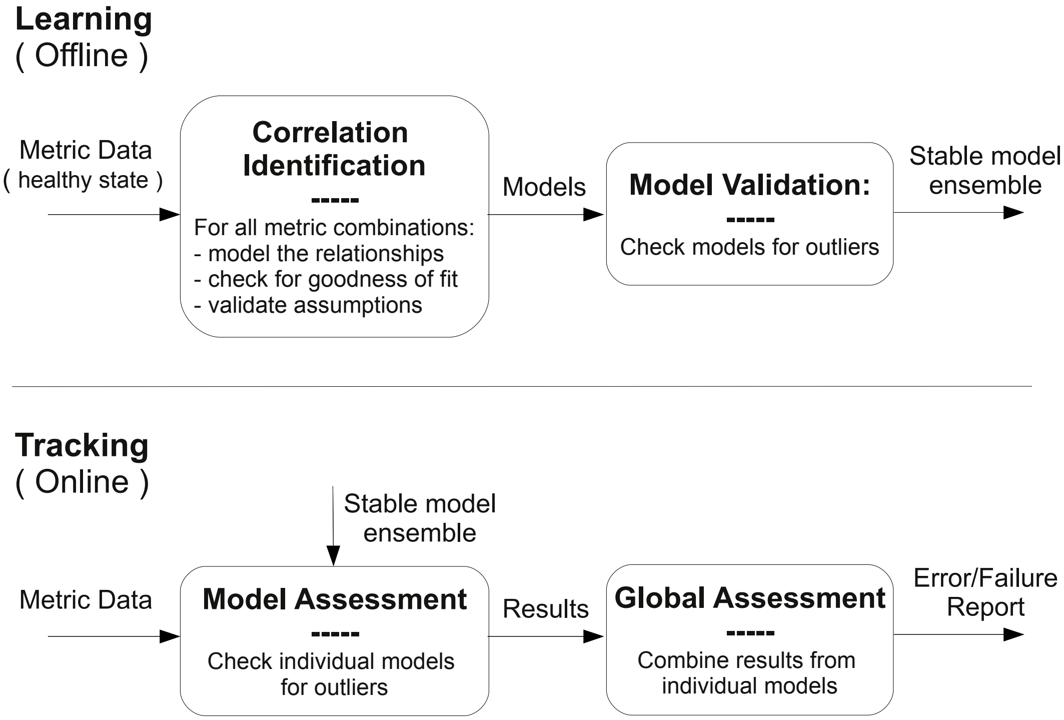


Figure 7.5: System modeling and tracking workflow

### 7.2.1 Correlation Identification

Let the number of metrics exposed by our target system be $n$ and the set of samples collected to build our system model be $S$. We split the set of samples $S$ into two sample sets $S_1$ and $S_2$, where $S = S_1 \cup S_2$. We evaluate all combinations of the $n$ metrics, by first estimating the model parameters using $S_1$ and then assessing how well the models fit the data in $S_2$.

The selection process first entails verifying any assumptions of the correlation models; models that violate the assumptions are discarded. The second step consists of measuring how well the model represents the underlying data. A number of measures are avail for this purpose. We measure the goodness of fit of a model by using the *coefficient of determination*, $R^2$, as it can be computed efficiently and is used widely in practice with regression models.

### 7.2.2 Model Validation

The set of samples $S_2$ is used to check whether the predictions of the remaining models are in line with actual observations. We consider a model to be valid if at at least $p_{min}$ of the samples available are within the acceptance bounds of the model ($p_{min} \in [0, 1)$). As such, only the models for which the number of outliers is less than $(1 - p_{min}) \times |S_2|$ are retained for use in system monitoring; the remaining models are discarded.

### 7.2.3 Simple Linear Regression

Many linear relationships exist between management metrics because of the underlying system structure. Certain system functions are dependent on others. For example, in an online store, a checkout is required to complete a purchase, making the number of purchases and checkout operations linearly correlated. Likewise, the time needed to display the results of a user query for a product may be linearly correlated with the time taken to execute the associated query on the back-end database. Simple linear regression (SLR) allows such linear relationships to be represented concisely. We cannot capture all pairwise metric correlations that exist in the system using SLR, as not all relationships are linear. However, as we show later, using SLR allows us to achieve good coverage of the available metrics.

#### Basics

SLR captures the relationship between two variables by fitting a line to the observed samples. An example of an SLR model is depicted in Figure 7.6. Given a set of pairs of values $\{(x_i, y_i)\}$, the learned model is given by:

$$\hat{y} = b_0 + b_1 x \tag{7.1}$$

where $\hat{y}$ is the estimated value of the dependent or predicted variable, the $b_j$'s are the model parameters, and $x$ is the independent or predictor variable. One method for finding the model parameter is Ordinary Least Squares (OLS), which involves minimizing the sum of squared residuals, *i.e.*, $\sum (y_i - \hat{y}_i)^2$. The solution to this optimization has a closed form and thus can be computed very efficiently. An SLR model can be computed in O($s$) where $s$ is the sample size used to estimate the model parameters.

#### Goodness of Fit

How well a model represents the data can be measured using the *coefficient of determination* $R^2$, which represents the proportion of the total variance in the

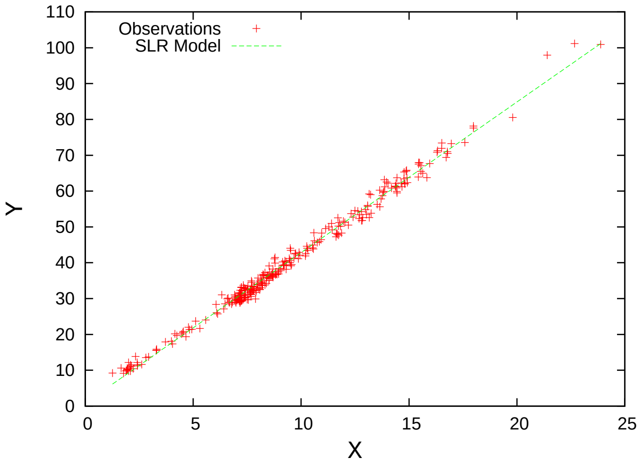


Figure 7.6: A linear relationship modeled by simple linear regression

dependent variable that is shared by the independent variable(s) in the model. The higher the $R^2$, the better the predictability of $y$ using $x$ as the predictor. $R^2$ is given by:

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

where $SS_{err}$ is the sum of errors squared, $SS_{total}$ is the total sum of squares, $\hat{y}_i$ is the predicted value, and $y_i$ and $\bar{y}$ are the observed and the mean values for the predicted variable. We are interested in models which have a minimum predictive accuracy. Thus, we keep a model if $R^2 > R^2_{min}$.

### Assumptions

Parameter estimation using ordinary least squares (OLS) assumes that the relationship between the variables is linear and that the residuals are independent and identically, normally distributed (i.i.d.). The latter assumption implies the following:

- The residuals ($i.e.$, $\hat{y}_i - y_i$) are independent of each other ($i.e.$, there is no serial correlation in the residuals), and they have a zero mean.
- The variance of the residuals is constant. The absence of constant variance is known as *heteroscedasticity*.
- The residuals follow a normal distribution.

These assumptions are typically verified by visualizing the data (*e.g.*, scatter plot of the observations versus predictions, scatter plot of the residuals versus observations, probability plot, *etc.*). This approach is not practical given the large number of metric combinations considered and the requirement to reduce reliance on human operators. Since our aim is to learn metric correlation models automatically, we make use of statistical tests to verify the assumptions.

The assumptions underlying OLS are not independent; often, failure to meet one assumption is accompanied with violations of others. For example, failure to meet the linear relationship condition is often detected by the fact that the residuals do not follow a normal distribution [28]. Similarly, the presence of residuals with non-constant variance may indicate a mis-specified model.

Failure to meet the assumption of independent residuals still produces unbiased regression coefficients (*i.e.*, there is no under or overestimation of the true parameter in the long run), but significance tests and confidence intervals are no longer valid. To check for independence of residuals, we apply an autocorrelation test; we test to see if residuals obtained with a model are correlated with a time-shifted version of itself. In particular, we employ the Durbin-Watson test [28], which checks whether the autocorrelation of residuals at lag one is zero.

Failure to meet the assumption of constant variance of residuals is generally seen as more important than failure of the other assumptions. Heteroscedasticity make significance tests and confidence intervals invalid. However, the estimated regression coefficients are still unbiased [28]. Also, Cohen *et al.* [28] suggests that heteroscedasticity has to be large for significance tests and confidence intervals to become incorrect. Several tests have been developed to detect if the variance of residuals is not constant and/or has a specific pattern. These include the *Goldfeld-Quandt* test, *Cook and Weisberg*, *Breusch and Pagan*, *White*, and Hartley's $F_{max}$ tests. Details on these tests can be found in [30, 100, 145, 147]. We use Hartley's $F_{max}$ test to check whether the residuals' variance has large differences at different values of the independent variable and at different points in time.

The assumptions underlying linear regression are not equally important. For example, failure to meet the normality assumption for the residuals does not invalidate the regression results, especially when the sample size is large [28]. Statistical tests such as *Anderson-Darling*, *Shapiro-Wilk*, and *Kolmogorov-Smirnov* (see [113]) can be used to see if a sample of observations comes from the normal distribution. However, because the sample sizes we consider to learn correlation models are large, we do not employ any test to check for normality. We found that these tests are too stringent and tend to prevent useful correlations from being retained. For example, we found many correlations that were stable but displayed a small level of heteroscedasticity; however, the use of normality tests failed on these correlations.

Violations of the OLS assumptions may be remedied by transforming the data. For example, the logarithm and square root transforms are known to help stabilize variance. An added benefit of transformation is that, often, addressing a departure from one of the assumptions causes the other assumptions to be satisfied [28, 113]. For example, a transformation may not only make a non-linear relationship linear, but also address problems of heteroscedasticity and non-normality. Later in this chapter, we discuss the use of SLR on transformed data to capture correlations for which SLR is not otherwise deemed to be a good fit.

### Outlier Detection

Once a metric correlation is identified and the corresponding model created, we need a way to check if new observations fit the learned model. The prediction error or residual is the discrepancy between the observed value for the dependent variable and the value predicted by the regression model; this is the basic ingredient we use to gauge how unusual observations are. In particular, we are interested in detecting outliers. An outlier is an observation that is not in line with what would be expected from the assumed correlation.

A number of outlier detection statistics have been described in the literature. We use the *studentized residual*, which represents the difference between model prediction and actual observation when a new observation is not included in the estimation of the model parameters. Two variants of studentized residuals exist: internally studentized and externally studentized. We use the latter as it is the preferred variant for outlier detection [28]. The externally studentized residual is computed as follows:

$$d_i = \frac{y_i - \hat{y}_{(i)}}{\sqrt{s_{\epsilon(i)}\sqrt{1-h_i}}} \tag{7.2}$$

where $h_i$, the leverage of the $i$th point, is computed as follows:

$$h_i = \frac{1}{n} + \frac{(x_i - \overline{x})^2}{(n-1){s_x}^2}$$

$s_{\epsilon(i)}$ is the standard deviation of the residuals from the model computed without using the $i$th observation, $\hat{y}_{(i)}$ is the predicted value without using the $i$th observation in the model, $p$ is the number of parameters of the model including the intercept, $n$ is the number of observations, and $s_x$ is the standard deviation of the independent term.

A model detects an outlier if $|d_i| > D_{max}$. Guidelines exist on how to set the value of $D_{max}$. When a regression model fits the data, the externally studentized residuals are expected to follow a t-distribution with $df = n - p - 1$ [28]. When the sample used to learn the model is large, cut-off values as large as ±4.0 can be used [28]. We adopt the latter recommendation in our work.

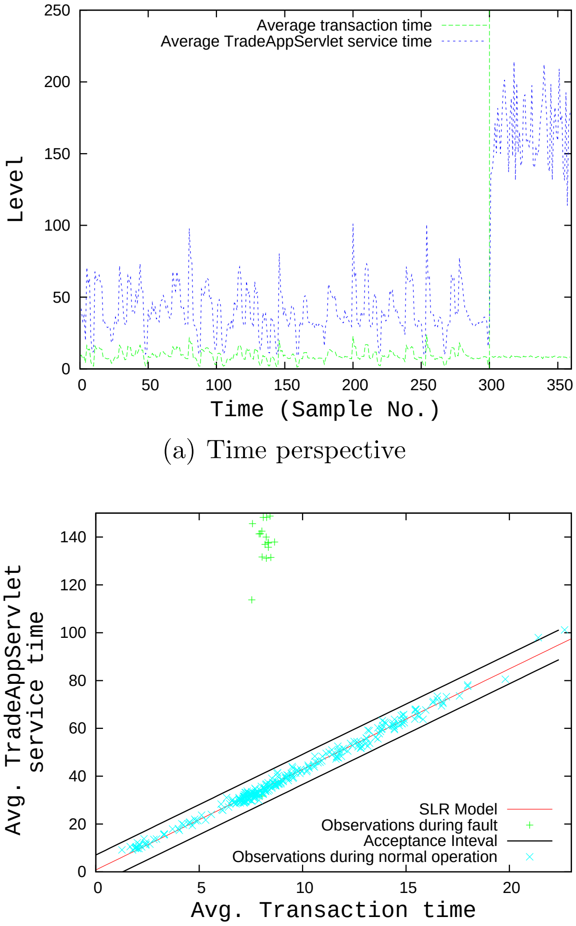
Figure 7.7 depicts an example of a metric correlation, which is taken from our experimental data. The behaviour of the two metrics is shown both as a function of time and as a function of each other. Figure 7.7(b) shows the linear regression model that captures the correlation as well as the prediction interval that delineates normal observations from outliers. The figures also illustrate the effect of a fault on the correlation. At some point, a fault is activated in the system and this particular correlation model clearly detects a number of outliers.

For monitoring purposes, one limitation of the studentized residual is that it is sensitive to the standard deviation of the residuals (*i.e.*, the term $s_{\epsilon(i)}$). For very strongly-correlated pairs of metrics, this value is very small. As a result, it is possible for slight discrepancies between predictions and observations to be seen as significant, even though the absolute discrepancies are not material. A discrepancy of 1 ms in a response time metric whose mean value is 100 ms is not material. However, such a change may be significant statistically if the metric was involved in a strong correlation.

To address this problem, in addition to using the studentized residual, we also compute the relative absolute residual, $r_i$, which is given by:

$$r_i = \frac{y_i - \hat{y}_i}{\hat{y}_i} \tag{7.3}$$

For an observation to be considered anomalous, we require both $|d_i| > D_{max}$ and $|r_i| > E_{max}$. The value of $E_{max}$ depends on what represents material discrepancy, which is system-specific.



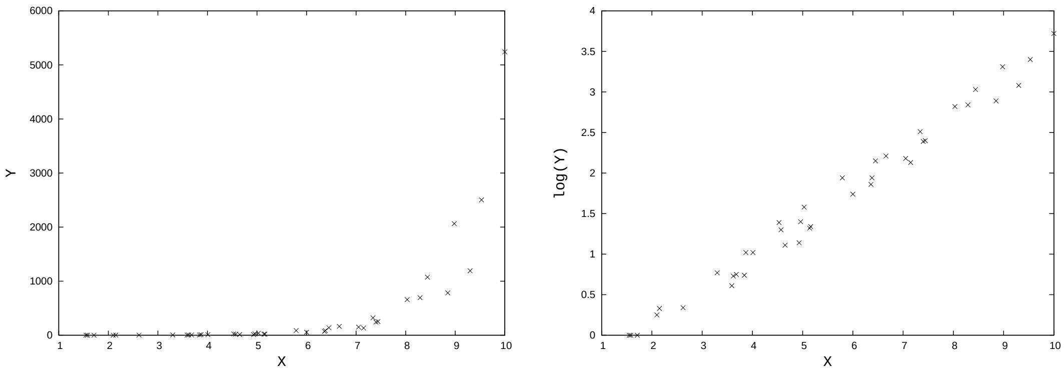
(a) Time perspective

(b) Correlation perspective

Figure 7.7: Sample fault: Effect on a correlated metric pair

### 7.2.4 Extensions and Variations

Not all relationships in software systems can be captured using SLR; many relationships are non-linear, reflecting the complexity and dynamic nature of software systems. For example, the relationship between load and response-time is not linear. When a system reaches saturation, response time increases exponentially. Similarly, many constraints and bottlenecks that exist in a software system give rise to non-linear behaviour. We investigate a set of modeling techniques that can represent non-linear behaviour. Our choice of techniques is driven by the need to keep the modeling cost low and to capture different kinds of behaviour we may encounter in practice. These modeling techniques are variants and extensions of linear regression.



(a) Raw data

(b) Data after applying a logarithm transform

Figure 7.8: Applying data transformation: An example

### SLR with Transformed Data

To capture common forms of non-linear behaviour and to address other peculiarities of the data, we transform the data as a preparatory step and, then, apply SLR. Models based on SLR with transformed data (SLR-T) are of the following form:

$$T(y) = b_0 + b_1 T'(x) \tag{7.4}$$

where $T(.)$ and $T'(.)$ can be any transformation used in data analysis.

Data transformation is commonly used to linearize a non-linear relationship, to reduce the heterogeneity of variance, and/or normalize model variables. An example of linearization by means of a logarithm transform is presented in Figure 7.8.

The traditional approach is to inspect the joint distributions and residuals manually, for all pairs of variables considered, in order to decide what transformation is required. However, this approach is impractical for our purposes. We thus automate the process by selecting the transformation that produces the most accurate predictions while satisfying OLS assumptions. We use $R^2$ to identify the best transformation. It should be noted that we compute $R^2$ based on the residuals computed in the original units; that is, we reverse any transformation applied to the predicted values before computing $R^2$.

### SLR with Smoothed Data

SLR with Smoothed Data (SLR-S) entails smoothing the data prior to learning a model. Smoothing reduces noise in the data by dampening changes between successive data points. Even though smoothing may cause subtle discrepancies due to faults to be hidden, we expect to discover more correlations, which should improve the ability to detect anomalies in the system.

An SLR-S model has the following form:

$$S(y) = b_0 + b_1 S(x) \tag{7.5}$$

where $S(.)$ is the smoothing function. In this work we use an unweighted sliding-average smooth whereby each point is replaced by the average from the previous $k$ samples. The smoothed value is given by:

$$S(x_i) = \frac{1}{k} \sum_{j=i-k+1}^{i} x_j$$

### Autoregressive Models with eXogenous Input (ARX)

An Autoregressive model with exogenous input (ARX) [85] can be described as a linear difference equation. Such a model can capture correlation between variables which display serial correlation. In addition, such a model can take advantage of the recent trends of the variables to improve prediction.

An ARX model based on two variables but with additional lagged terms has the form:

$$\begin{aligned} y(t) &= a_1 y(t-1) + ... + a_k y(t-k) \\ &\quad +b + b_0 x(t) + ... + b_l x(t-l) \end{aligned} \tag{7.6}$$

where $y(t)$ and $x(t)$ are the input and output at time $t$ and $(k, l)$ represent the maximum time lag of the two variables relevant to the model. The dependency on the lagged versions of $y$ is the autoregressive part of the model and the $x$'s represent the exogenous or extra variable. We should note that time is discretized and that $t$ varies by one unit for each sampling interval.

To determine the best values for $k$ and $l$, we compute all models such that $k \leq L_{max}$, $l \leq L_{max}$ and select the best among them using the *adjusted* $R^2$ ($\bar{R}^2$). We do not use $R^2$ directly because it can only increase as more terms are added to the model. Adding more variables, however, reduces the parsimony of the model. Instead, we use $\bar{R}^2$, which includes a penalty for each new term added to the model.

It is given by:

$$\bar{R^2} = 1 - (1 - R^2)\frac{n-1}{n-p-1}$$

where $n$ is the number of data points and $p$ represents the number of independent terms in the model.

### Locally-Weighted Regression

Patterns of non-linear behaviour can be complex; it is impossible to consider all such patterns. In this work we use locally-weighted regression (LWR) (see [9]) to capture non-linear behaviour that our other techniques may not be able to capture. In particular, LWR obviates the need to specify a global function to model the data;

LWR works by fitting a local linear model for each prediction by emphasizing nearby data points. Note that with this technique the learning data has to be kept and is used when a prediction needs to be made. The local model is obtained by minimizing the locally weighted sum of the squared residuals (*i.e.*, $\sum {w_i}^2(y_i - \hat{y}_i)^2$) for all points in the learning data. The weights are given by:

$$w_i = K(\frac{D(x_i, x_{query})}{h})$$

where $K(.)$ is the weighting function, $x_{query}$ is the independent value for which a prediction is needed, $D(.)$ is the distance, and $h$ is the *kernel width*. $h$ is a parameter that determines how distant points affect the local model; the larger it is the smoother is the prediction function. Different ways of choosing the kernel width have been proposed [9], including recommendation for default values [113]. In this work we use the nearest-neighbour-based bandwidth selection, which consists of setting $h$ to the distance to $k$th closest neighbour.

An example of a relationship modeled using LWR is presented in Figure 7.9. We can see that the model is more flexible than SLR-T, as it takes account of the local distribution of the observations.

## 7.3 Suitability for Adaptive Monitoring

We can use a metric correlation model to check the behaviour of the associated metrics. Each model can be evaluated whenever the data it requires becomes

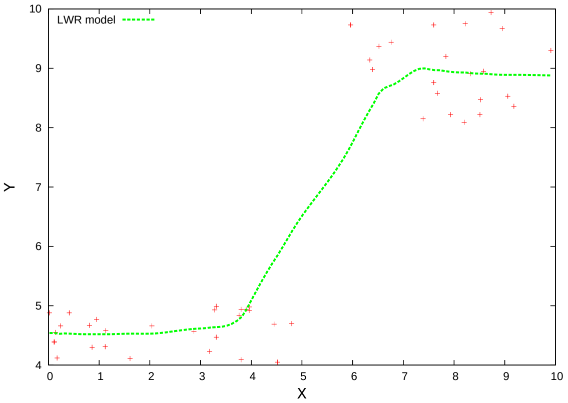


Figure 7.9: A relationship modeled by locally-weighted regression

available. This makes our system model, which is an ensemble of metric correlation models, most fitting for adaptive monitoring. At any point in time, we only need to employ the models whose metrics are being collected.

In order to track the system health at low cost, we can track a subset of the available metrics continuously through the associated models. The remaining models can then be used as needed (*e.g.*, when faults are suspected). This idea forms the basis of our adaptive monitoring approach, which is discussed in depth in Chapter 8.

## 7.4 Experiments and Analysis

### 7.4.1 Data for Model Learning

In order to identify stable metric correlations for the Trade application, we use data collected over a period of approximately 18 hours, during which we ensured the system was in the healthy state. For the other applications, we use data from a roughly six-hour run to identify the correlations. Table 7.1 lists the parameter values we have used. Note that the metrics are collected using a 10-second sampling interval.

### 7.4.2 Calibration for Model Identification and Cost

Before applying the regression techniques discussed in this chapter, a number of parameters need to be set. In setting these parameters, we use recommendations from the literature, make use of very basic domain knowledge, and/or carry out a

| *Parameter* | *Values* | *Description* |
|---|---|---|
| $\lvert S\rvert$, $\lvert S_1\rvert$, $\lvert S_2\rvert$ (Trade) | 6300, 4200, 6200 | Model identification:<br># samples available for learning,<br># samples used for correlation identification,<br># samples used for model validation |
| $\lvert S\rvert$, $\lvert S_1\rvert$, $\lvert S_2\rvert$ (Other app.) | 2100, 1400, 2100 | Same as above |
| $p_{min}$ | 0.99 | Model validation:<br>Min. proportion of samples required to be valid |

Table 7.1: Parameters used to compute and validate correlation models

search for appropriate values. When searching for parameter values, we constrain the search space to keep the computational cost low and maintain the practicality of our approach.

As the number of possible models is large, the modeling techniques we use have to be efficient. We next discuss how we configure the modeling techniques for use in our approach, and we analyze their cost.

### SLR

To check the assumption of constant error variance, the typical approach is to consult an F-table with a given significance level and the sample size. However, Cohen *et al.* [28] suggests that if the ratio of conditional variances at different values of the independent variable exceeds 10, heteroscedasticity can be considered to be large, and thus problematic. We take a more conservative approach than this rule of thumb. When heteroscedasticity is a function of time, we use the critical value from the F-distribution with a significance level of 0.001 (*e.g.*, $F_{\alpha=0.001,500,500} = 1.32$). Otherwise, we discard models for which $F_{max}$ exceeds 4.0. Our filter rule is more stringent for time-induced heteroscedasticity because models which suffer from this problem will most likely fail eventually, causing unnecessary false alarms. In the case where heteroscedasticity is induced by the independent variable, we can be more lenient, as the data used to learn the models captures the expected range of the variance.

To detect outliers during model validation and monitoring, we set $D_{max} = 4.0$ and $E_{max} = 0.2$. The same parameter values are used to detect outliers with SLR-T and SLR-S.

Among the techniques evaluated in this work, SLR is the most computationally efficient. Each model can be computed in O($s$), where $s$ is sample size used for learning. The cost of making a prediction with an SLR model is O(1).

### SLR-T

We limit ourselves to common forms of data transformations; these are listed in Table 7.2. These four transformation functions generate 16 possible models for each pair of variables considered.

| *Transform* | *Equation* |
|---|---|
| None | $T(x) = x$ |
| Logarithm | $T(x) = log_{10}(1 + x)$ |
| Inverse | $T(x) = \frac{1}{1+x}$ |
| Square root | $T(x) = \sqrt{x}$ |

Table 7.2: Data transformations considered

SLR-T and SLR-S are in effect SLR models applied to transformed or smoothed data. Like SLR, each model can be computed in O($s$), even though the complexity for computing SLR-S and SLR-T models is a multiple of SLR. This multiple is 16 for SLR-T since we consider four possible transformations for each variable.

### SLR-S

For each pair of variables considered, we search for the best value of the smoothing parameter $k$ from the candidate set $\{1, 3, 6, 9, 12\}$. The best model is the one having the highest $R^2$. We do not consider values of $k$ greater than 12 because of the delay it would introduce for error detection. We elaborate on mechanisms for error detection in Chapter 8.

Because we consider five different cases for smoothing the metric values, the cost is 25 times that of SLR. However, the cost is still O($s$).

### ARX

We solve ARX models by treating them as multiple linear regression models. Let Y be the $s \times 1$ vector of observed dependent variables and X be the $s \times p$ data matrix with all the predictor terms and one additional term to cater for the intercept; $s$, here, is the size of the sample used to compute the models. Deriving the model parameters requires computing $[\mathbf{X'X}]^{-1}\mathbf{X'Y}$. This equation is usually solved by QR decomposition of X with complexity O($sp^2$). In addition to these computations, we need to search for the best lag for each of the two variables in a model. In our evaluation, we set $L_{max}$ to 2, as user requests in our test-bed and similar systems are of short duration and do not span the allowed three sampling intervals (*i.e.*, 30 seconds). For each pair of variables, we thus have 9 possible models, contributing a factor of 9 to the cost of model identification.

Note that to compute the studentized residuals in the case of ARX, we would need to compute the leverage (*i.e.*, $h_i = \mathbf{x_i}[\mathbf{X'X}]^{-1}\mathbf{x_i}')$ where $\mathbf{x_i}'$ is a $1 \times p$ vector representing the sample being tested. However, this operation is expensive, as it usually involves a QR decomposition. In our work we resort to using standardized residuals to check for outliers, which can be computed with cost O(1). Standardized residuals can be computed thus:

$$d_i' = \frac{y_i - \hat{y}_{i(i)}}{\sqrt{s_{\epsilon(i)}}}$$

We set the maximum residual based on $q^{th}$-percentile residual observed during model validation. If a newly observed sample produces a residual that exceeds the percentile value by a factor $\gamma$, we consider it to be an outlier. In our evaluation, $q$ is set to 95% and $\gamma = 1.5$.

### LWR

We use the tricube kernel as the weighting function, which determines weights of neighbouring points as a function of distance. The kernel function needs to be smooth, but the choice of the function is not a critical one [9]. Also, for each pair of metrics, we consider three values of $k$ to identify the $k$th nearest neighbour, namely $\frac{s}{8}$, $\frac{s}{4}$, $\frac{3s}{8}$,$\frac{s}{2}$, where $s$ is the sample size used to the learn the model. From these, we choose the model that yields the highest $R^2$ value.

Of the modeling techniques we consider, LWR is the most computationally expensive. Because it is a lazy-learning technique, no model parameters are estimated at the time of learning. Instead, all points collected during learning are kept in a data structure, and used only when a prediction needs to be made. At the time of prediction, we need to find the $k$-nearest neighbours, and then, we need to use them to fit a local regression and make a prediction. If we use KD-Trees for nearest neighbourhood search, learning amounts to constructing the trees with cost O($slog(s)$). With $k$ neighbours, the cost of a single prediction is approximately O($k^2$log($s$)). In the implementation we use, all neighours (*i.e.*, $s$) are taken into account for prediction; the cost is thus O($s^2$log($s$)).

The computation and memory requirements for LWR are a function of the number of samples used to learn the model. Using all the data we have available for correlation identification would be too costly. Instead, we use 180 samples (corresponding to three hours) for learning LWR models. Nevertheless, we use as many samples to validate the models as the other modeling techniques.

The outlier detection technique presented in Section 7.2.3 does not readily extend to LWR. As such, we do not use the studentized residuals to detect outliers. Instead, we use the same approach and parameter values as ARX models to identify outliers.

### 7.4.3 Setting $R^2_{min}$

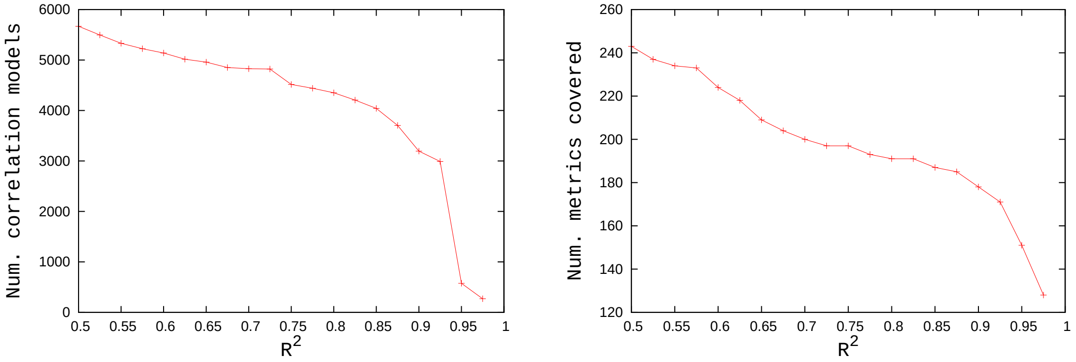
One important parameter of correlation identification is $R^2_{min}$. In choosing an appropriate value for $R^2_{min}$, we are faced with two conflicting requirements. On the one hand, we wish to retain as many correlations as possible to be more sensitive to disturbance in the system; this can be achieved by choosing a small $R^2_{min}$ value. On the other hand, we would like to keep only those metric correlations that fit the data well so as to not miss anomalies and avoid false alarms; this requires larger $R^2_{min}$ values.

Figure 7.10 shows the effect of computing SLR models with different $R^2_{min}$ values for the Trade system. As expected, we observe that higher $R^2_{min}$ lead to the identification of fewer correlations and the coverage of a smaller subset of the Trade system metrics.

Details of our error detection approach are provided in Chapter 8. We nevertheless present some results to give some intuition on the effect of varying $R^2_{min}$. Figure 7.11 presents a summary of the results of monitoring the Trade system using SLR models. These results indicate that for the set of faults we use the choice of $R^2_{min}$ is not sensitive until we reach a value of 0.875; at that point, our ability to detect fault-induced disturbance decreases while we fare better in avoiding false alarms. The choice of 0.875 would be adequate if we knew the faults in advance and were confident that these faults are representative of the faults the system can experience. Both arguments are not applicable since we have assume no prior knowledge of faults.

In our evaluation we set $R^2_{min}$ to 0.6, which means that the models have better than average predictability; each model can explain at least 60% of the variability in the predicted variable. With this choice, we can not only capture strong metric correlations but also retain weaker but stable correlations. Figure 7.11 indicates that a significant number of false alarms are raised when $R^2_{min} = 0.6$, we later show that these false arise because of only a few models.

Although we are lax in setting $R^2_{min}$, our validation step is more strict in checking the stability of the models. In the validation step we discard the models which detect outliers without there being a fault. More specifically, we require each model to be valid for 99% of observations used for validation (*i.e.*, we set $p_{min} = 0.99$).



(a) Effect of varying $R^2_{min}$ on model identification

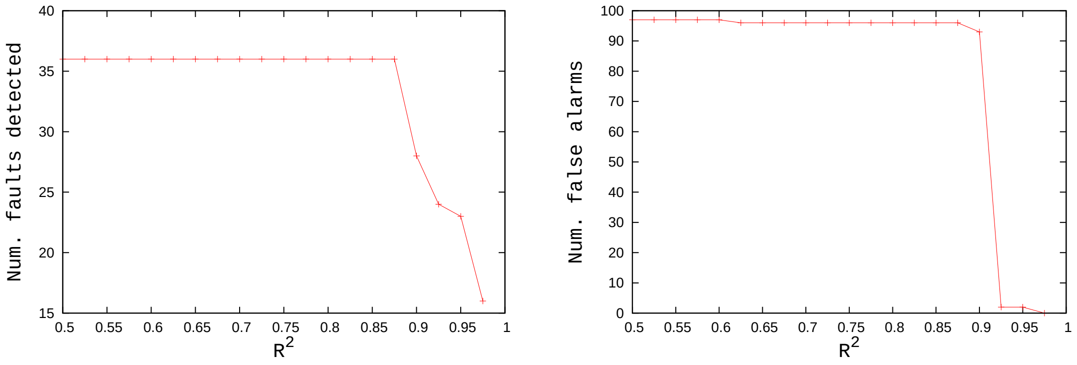
(b) Effect of varying $R^2_{min}$ on metric coverage

Figure 7.10: Correlation models and metric coverage

### 7.4.4 Existence of Stable Metric Correlations

To show that stable correlations between metrics exist, we apply the regression modeling techniques to data collected from out test-bed. In our experiments, Trade and PlantsByWebSphere are subjected to a random uniform workload, while for RUBiS and TPC-W we use the default workload generated by the emulated clients as provided in the package.

Table 7.3 summarizes the results of our modeling approach using SLR. The first two columns indicate the number of metrics we use for modeling and the number of components the metrics cover. The next three columns in order list the number of models identified, the number of metrics these models cover, and the number of components covered. We first observe that with the exception of TPC-W half



(a) Effect of varying $R^2_{min}$ on the ability to detect faults

(b) Effect of varying $R^2_{min}$ on false alarms

Figure 7.11: Sensitivity to faults

or more of the components considered are associated with metric correlations. For Trade, which is our main target application, the models cover more than 60% of the considered metrics and more than 87% of the components. Despite the fact that SLR can only capture linear correlations, these results indicate that we can track the health of a major part of the system.

The configuration and structure of a system determine what types of relationships exist between metrics. For example, a saturated tier in a system may cause many relationships, which otherwise would be linear, to become non-linear. As such, we may not be able to achieve much metric and component coverage with SLR. The RUBiS and TPC-W applications suffer from this problem; for these applications, the database tier is a bottleneck. In such cases, we can use variants of SLR to improve the system coverage and, hopefully, the ability to detect errors. To illustrate this, we apply SLR-T to the data collected from our test-bed. The results, shown in Table 7.4, show a marked improvement in metric and component coverage for all the applications. With SLR-T, we can cover more than half of the components of TPC-W, which has almost no coverage with SLR.

| *Application* | *# Metrics analyzed* | *# Components* | *# Models identified* | *# Metrics covered* | *# Components covered* |
|---|---|---|---|---|---|
| Trade | 352 | 40 | 5138 | 224 | 35 |
| PlantsByWebSphere | 269 | 36 | 199 | 109 | 19 |
| RUBiS | 59 | 20 | 122 | 31 | 10 |
| TPC-W | 51 | 20 | 1 | 2 | 1 |

Table 7.3: SLR modeling results

| *Application* | *# Metrics analyzed* | *# Components* | *# Models identified* | *# Metrics covered* | *# Components covered* |
|---|---|---|---|---|---|
| Trade | 352 | 40 | 17142 | 259 | 37 |
| PlantsByWebSphere | 269 | 36 | 1104 | 168 | 26 |
| RUBiS | 59 | 20 | 213 | 32 | 10 |
| TPC-W | 51 | 20 | 40 | 20 | 11 |

Table 7.4: SLR-T modeling results

Table 7.5 compares the coverage of the Trade application using the modeling techniques presented earlier. These results show that coverage can be improved significantly by using more-powerful techniques. In addition, we see that each of the alternatives to SLR allows a set of unique correlations to be found. However, as discussed earlier, these more-powerful techniques are costlier than SLR. If more coverage of the system metrics than that provided by SLR is desired, the other modeling techniques can be used, albeit incurring higher cost.

| *Modeling technique* | *# Models identified* | *# Unique models* | *# Metrics covered* | *# Components covered* |
|---|---|---|---|---|
| SLR | 5138 | 0 | 224 | 35 |
| SLR-T | 17142 | 152 | 259 | 37 |
| SLR-S | 24337 | 3104 | 261 | 37 |
| ARX | 18104 | 6939 | 351 | 40 |
| LWR | 21606 | 1000 | 284 | 37 |

Table 7.5: Metric correlation models from the Trade system

In order to understand the advantage provided by the modeling techniques we have considered, we classify the available metrics into three broad categories: *activity metrics*, *timing metrics*, and *state metrics*, and categorize the correlation models accordingly. Activity metrics are counters measuring the amount of work or number of operations performed in a given sampling interval. Examples include request counts, number of table rows selected, number of connections retrieved from pool, number of objects created, *etc.*Timing metrics mostly include response time metrics such as servlet response time, remote method response time, JDBC query execution time, *etc.* State metrics capture the current state of the system. Examples include memory and CPU usage, connection pool size, number of concurrent connections, *etc.*

Figure 7.12 shows the breakdown of the models with respect to our metric categories. This figure shows that most metric correlations captured by SLR involve activity metrics only. In contrast, the other modeling techniques are able to capture a significant number of relationships between different types of metrics, in particular between activity and timing metrics, and activity and state metrics.

Our modeling techniques rely on different features to improve on SLR. By smoothing, SLR-S is able to reduce the local variance of the metrics and thus increase the likelihood of finding linear correlations. The remaining techniques can capture non-linear correlations. SLR-T can represent basic non-linear relationships by using transformed data. In addition, transformation can address problems such as heteroscedasticity. SLR models that suffer from heteroscedasticity are discarded even though they display a high degree of correlation. With SLR-T, however, we can often stabilize the residual variance and thus retain some of the models that are discarded with SLR. LWR allows modeling of non-linear behaviour by leveraging data points close to those for which predictions need to be made. ARX models can capture non-linear relationships by taking advantage of the recent past of the predicted variable in addition to values of the independent variable. Figure 7.12 indicates that all alternatives to SLR are able to capture correlations between activity and timing metrics. ARX captures the largest number of such correlations, suggesting that the recent trend of the modeled metrics can help improve modeling of timing and activity metrics. Figure 7.12 also suggests that to capture metric correlations involving state metrics, we need to take account of the past behaviour. We observe that both SLR-S and ARX capture many such correlations. This matches the intuition that changes in state metrics are a reflection of the amount of activity that has taken place in an interval. It is harder for SLR-T and LWR to model these correlations, as they do not take account of past metric values.

Note that it is possible for the same pair of metrics to be captured by different modeling techniques. In our experiments, we have observed many cases where a pair of metrics is modeled differently by different techniques. Since we do not interpret the models, they do not have to accurately and intuitively reflect a phenomenon in the system. As long as the model has good fit and satisfies our quality requirements, we can use it for system monitoring.

### 7.4.5 Error Detection with Metric Correlations

We now take a look at how the modeling techniques compare in capturing correlations that are effective for monitoring a system. Specifically, we study how effective

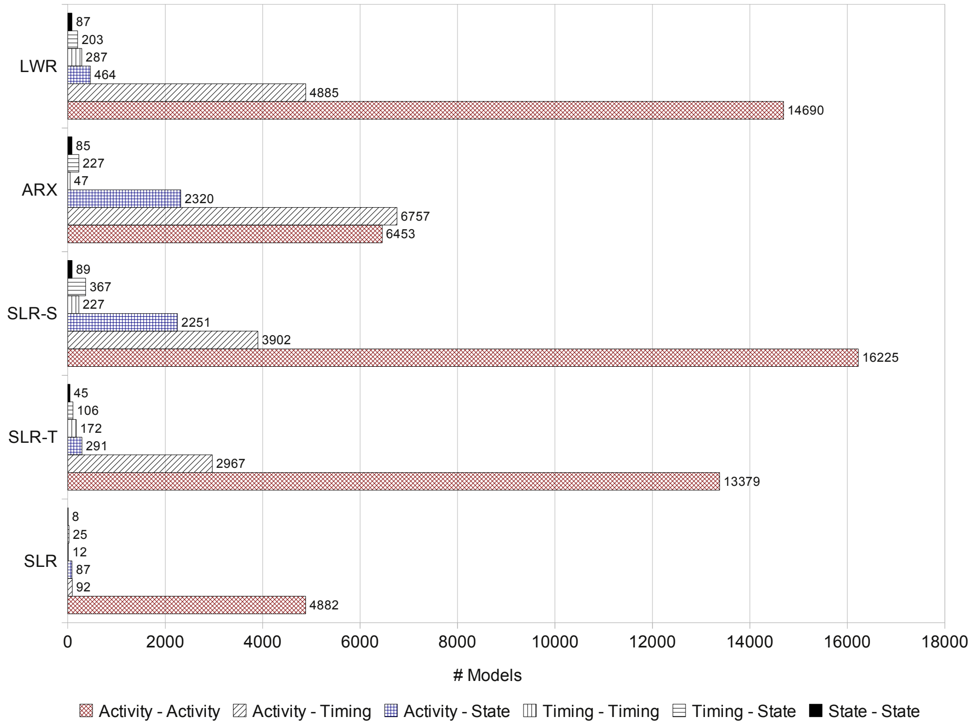


Figure 7.12: Comparison of modeling techniques per type of metric pairs covered

the correlation models are when all of them are used for monitoring the Trade system. For specific details of our error detection approach, the reader is referred to Chapter 8.

Table 7.6 shows that the number of faults detected by the different techniques is roughly the same. SLR-S detects the maximum number of faults, followed by LWR, which detects one less than the maximum. These two techniques generate the highest number of correlation models (see Table 7.5). We also used 18-hour long data collected from the system in the healthy state and apply the same correlation models. We report the number of false alarms (*i.e.*, the number of times anomalies are reported, even though no fault was injected) and the number of unique models that cause those false alarms.

From these results we observe that there is only a small difference in fault detection using the different modeling techniques, despite significant differences in the number of models identified. For example, the fault coverage of ARX is the same as SLR despite ARX having more than three-fold the number of models and higher metric and component coverage. This indicates that there may be much redundancy in the coverage of the more-powerful techniques.

Table 7.6 also presents the false alarm results. The number of false alarms suffered by SLR may seem high, but these correspond to 10 models only. Note that we do not discard models that cause false alarms. In practice, however, these will be discarded, which would make the false alarms number much smaller.

Using the same amount of data and similar parameters, the other modeling techniques are less robust than SLR. SLR-S and LWR suffer from the highest number of false alarms. SLR-T also suffers from a relatively high number of false alarms. ARX fares better than the other alternatives to SLR, but is still worse than SLR. This lack of robustness is explained by the fact that the more-powerful techniques may overfit the data. When models lack the ability to generalize, valid observations may be detected as outliers.

| *Modeling technique* | *# Faults detected* | *# False alarms* | *# Distinct failed models* |
|---|---|---|---|
| SLR | 36 | 97 | 10 |
| SLR-T | 35 | 373 | 134 |
| SLR-S | 39 | 859 | 1241 |
| ARX | 36 | 121 | 46 |
| LWR | 38 | 428 | 106 |

Table 7.6: Comparison of fault coverage and false alarms

In the absence of knowledge about faults and the metrics which the faults could affect, it is better for our ensemble of models to cover the maximum number of metrics and components. The more components covered, the more likely we are to uncover faults in the system. Also, the broader the coverage, the more confidence we can have in the analysis, as it is based on a more-complete picture of the system. However, the need for more coverage should be balanced with the need to keep the memory and computation cost of learning and applying the models low. Furthermore, because false alarms require the attention of system operators, it is essential that the false alarm rate be minimized.

From our experience, we recommend the following approach to modeling metric correlations.

- Start with SLR. There are many benefits to using SLR. First, the modeling assumptions can be checked using established, well-studied statistical tests. Second, outliers can be detected by employing established, efficiently-computed diagnostics. Third, the models can be interpreted easily. The alternative modeling techniques can produce models that are difficult to interpret. For example, it can be difficult to interpret ARX models that have negative coefficients or certain models with transformed data (*e.g.*, using a square-root transform).

- If the coverage provided by SLR is not satisfactory, for each unmodeled pair of metrics, consider using ARX then SLR-T. However, both ARX and SLR-T should be used with stricter model selection and validation parameters. In particular, we recommend using a higher value of $R^2_{min}$ and more samples for model validation.

We do not recommend the use of SLR-S, for much experimentation would be needed to determine appropriate parameters to reduce false alarms to an acceptable level. We also do not recommend using LWR because of its high cost both in terms of memory and computation. In order to make this technique practical, we need to either reduce the size of the learning data or the number of metrics that need to be tracked, none of which is a satisfactory option. With less data, we may not be able to capture representative behaviour of the system.

## 7.5 Summary

In this chapter we have presented and evaluated an approach to modeling complex software systems based on the management metrics they expose. The main idea is that a system in the healthy state displays stable, time- and load-invariant correlations among many of its metrics; some of these correlations are perturbed when faults occur in the system. The metric correlations are formalized mathematically as regression models. Together, these models form an ensemble, which represents our system model. Our principal motivation for using the correlation-based modeling approach is that it is suitable for adaptive monitoring. Our system model enables tracking different subsets of the available metrics at different times.

We study and compare several practical regression techniques to capture metric correlations. We leverage standard, well-studied, statistical tests to identify stable correlation models and to detect outliers. Using a multi-tier software system, we show the existence of stable metric correlations in several Java EE application benchmarks. Our results indicate that a significant amount of metric and component coverage can be achieved using simple linear regression alone. We investigate the use more-powerful variants of linear regression to capture non-linear behaviour.

Our experiments show that these techniques not only improve metric and component coverage, but they also allow metrics of different nature to be correlated. However, these more-powerful techniques have higher computational cost and the models they produce tend to be more vulnerable to false alarms.

# Chapter 8

# Adaptive Monitoring

The cost of measuring and collecting all available metrics from a system is generally high and cannot be incurred permanently in a production software system. However, tracking metric behaviour is necessary for determining the health of the system and for performing fault localization. Fortunately, these two tasks do not have the same requirements in terms of the metrics that need to be tracked. Errors or failures are rarely confined to small parts of a system; instead, it is more common for faults to have widespread effect and/or for errors and failures to propagate in the system. As a result, a small set of metrics is often sufficient to estimate the health of the system. Localizing faults, on the other hand, necessitates acquiring a more-complete picture of the system, as a fault can exist anywhere in the system. Therefore, one effective approach to reducing the high cost of full metric monitoring is to only collect and analyze enough metrics to allow those two tasks to be carried out.

In this chapter we describe our automated, adaptive monitoring approach that allows the cost of monitoring to be kept low, while still providing pertinent data for fault detection and diagnosis. The adaptation of monitoring refers to the changes made at runtime to the set of metrics collected from the system. An adaptive monitoring system can, therefore, be viewed as a system which tracks different subsets of the available metrics at different times. This capability allows the system to control the monitoring overhead.

Our choice of system modeling approach as described in Chapter 7 is specifically motivated by the requirement to enable adaptive monitoring. In order to track different subsets of metrics at different times, we simply need to make use of the correlation models that are associated with the metrics in the subsets. This opens the door for different adaptation algorithms to be devised. As illustrated by the diagram in Figure 8.1, at any point in time these algorithms can choose to oversee any subset of the modeled metrics to meet their goals.

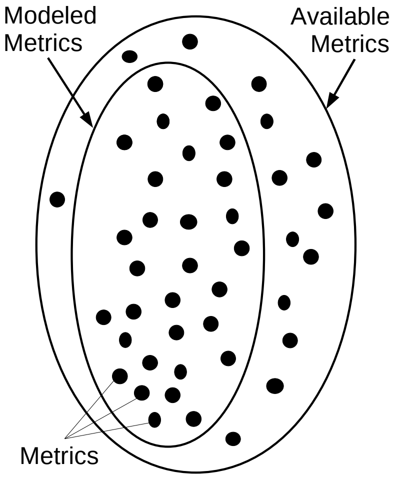


Figure 8.1: Available metrics and the subset of modeled metrics

Since we do not assume any knowledge about the target system's internal structure nor any faults that may occur in the system, we devise a basic approach to adaptive monitoring. In this approach, we pre-specify different monitoring levels. At each such level, a given set of metrics is collected and their behaviour analyzed. When metrics exhibit anomalous behaviour at one level, we move up to the next, more-detailed level to gain further insight. We next describe this approach with two levels of monitoring, namely *minimal* and *detailed*.

The purpose of detailed monitoring is to obtain a more-complete picture of the system health and to perform diagnosis. This level of monitoring is triggered when faults are suspected in the system, and the behaviour of metrics is tracked using the metric correlation models we have identified *a priori*. At this level, we incur the cost of collecting all the modeled metrics. However, this cost is incurred for short lapses of time; prolonged detailed monitoring is not necessary since we are not interested in studying any particular, evolving phenomenon. Rather, we only need to gather sufficient evidence to reason about possible faults and their location in the system.

In this work we presume that all metrics associated with correlation models are collected at the detailed monitoring level. However, even if we limit the collection to those metrics that are associated with models, the monitoring overhead can still

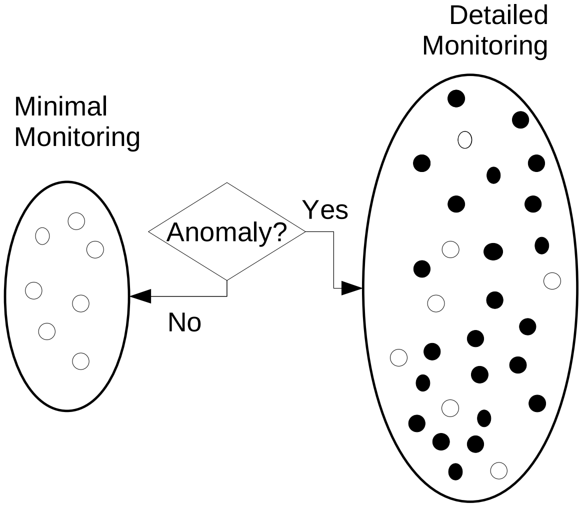


Figure 8.2: Adaptation in the context of two levels of monitoring

be high. If there are constraints on the cost of monitoring that system operators are willing to accept at the detailed monitoring level, we need to restrict the set of metrics collected. To this effect, we can select a subset of the modeled metrics whose measurement and collection overhead fits a system operator-specified budget. Following the cost estimation approach described in Chapter 6, this budget can be put in terms of the slowdown system operators are willing to tolerate in a system, which may have only failed partially. Later in this chapter we discuss methods to select subsets of metrics to monitor.

The goal of minimal monitoring is to provide a continuous, low-cost assessment of the overall system health. Cost reduction is achieved by limiting the collection to a small set of metrics. Minimal monitoring is essential in keeping the monitoring cost low, because it is in effect while the system is in the healthy state. A production system is expected to be in the healthy state most of the time, and faults are anticipated to occur rarely. When anomalies are detected at the minimal level, adaptation takes place, and a larger set of metrics is collected and the data analyzed. Figure 8.2 provides a high-level depiction of our approach to adaptive monitoring.

Metric selection is critical for the minimal monitoring level, for it needs to detect as many faults as possible and its cost is incurred continuously. We next present several methods for selecting a subset of metrics to collect at the minimal monitoring level. We then describe how the health of a system can be gauged at the two monitoring levels.

## 8.1 Metric Selection

In deciding what metrics to collect at a particular monitoring level, we need to take account of the following important factors:

- *Importance:* When domain knowledge is available, some metrics may be known to be more critical than others (*i.e.*, they reflect measures of interest to the system operators). Such selection, however, presumes that system operators understand the system and know which metrics reflect critical aspects of a system.
- *Cost:* Metric measurement and collection reduce system efficiency directly. The performance cost depends on what quantity is measured and how often it is measured. The higher the cost of a metric, the more careful we have to be in enabling its collection.
- *Redundancy:* A number of metrics may reflect the same underlying phenomenon. Also, many metrics may behave similarly. In such cases, it may not be necessary to collect all these metrics; most information from these metrics can be had through a set of selected representatives.

Based on the above factors, we consider two broad approaches to selecting metrics: *manual selection* by leveraging domain knowledge and *automated selection* by making use of information inferred from the monitoring data.

### 8.1.1 Manual Selection

With some domain knowledge, we may be able to identify critical indicators of a system's health. These indicators often include utilization of key resources (*e.g.*, CPU, memory, network), performance of major components of the system (*e.g.*, number of requests successfully serviced and response time), and number of failures encountered. It is also possible that the monitoring of some critical metrics is recommended by system vendors (see, *e.g.*, [62]).

In the absence of any prior knowledge and recommendations, certain metric characteristics make them better candidates than others for inclusion in the selected metric set. First, the metrics are affected by the behaviour of a large number of components, allowing them to detect a broad range of errors and failures. Second, these metrics are not expensive to measure and collect. Third, problems that do not affect the selected metrics are deemed not pressing enough to warrant immediate further investigation.

The difficulty with this selection approach is that it is manual, calling for human input. Its results will vary from one individual to the other, depending on their knowledge and experience.

### 8.1.2 Automated Selection

An alternative to relying on the knowledge of system operators is to determine automatically, by analyzing the data exposed by the system, a small set of metrics suitable for tracking the system's health. With this approach, we no longer rely on the metrics' semantics; instead, our choice is dictated by cost and information-redundancy considerations. Because of our focus on correlation-based monitoring, our selection methods produce metric subsets that can be tracked using metric correlation models. These methods use the metric correlation information as their main input. Note that we do not employ any threshold-based models when the minimal set of metrics is determined automatically.

Even though metric correlations do not correspond to dependencies in the system, they contain valuable hints about the system structure and its dynamics. The correlations reflect underlying artifacts such as component dependencies, component interactions, and workload-induced shared behaviour. While the global dynamic behaviour of a system is typically driven by the input workload, the system structure causes differentiation among metric correlations. For example, if one operation always causes another to be performed, metrics which reflect the frequency of the two operations will be strongly correlated. On the other hand, two operations that tend to vary with the workload without a direct dependency between them may be less strongly correlated. Such differentiation allows different dynamics in the system to be reflected in the correlations. Thus, the correlation information can help us decide on what metrics to choose.

The metric measurement and collection overhead can be viewed as comprising two parts: one that is constant (*e.g.*, connection setup between the data collector and the target) and one that varies with the number of metrics collected (*e.g.*, metric measurement, operating system overhead for transferring the metric values, *etc.*). By interpolating the performance overhead results in Chapter 6, we find that the constant component represents a small portion of the overall overhead. As such, it is reasonable to assume that the monitoring overhead is proportional to the number of metrics collected. In the following sections we discuss metric selection methods under the constraint that there is a maximum number of metrics which we are allowed to collect. This constraint is akin to a maximum performance overhead budget specified by system operators.

Note that the measurement overhead varies from one type of metric to another. For example, an activity counter is less expensive than a time-tracking metric; the latter involves two system calls for each update. However, with the exception of one method, all methods discussed in this work involve a random choice of metrics from some set of metrics. Therefore, we will obtain a mix of metrics of different types, and the chance of selecting only high-cost metrics is low.

### Naïve Selection

We first consider two simple and intuitive methods to select metrics.

- **Random selection:** This method entails selecting randomly from the set of pairs of correlated metrics until the maximum number of metrics desired is reached. This method allows a uniform coverage of the space of metric correlations.

- **Selection by strongest correlation (*i.e.*, minimum distance):** This method focuses on metric pairs that have the highest $R^2$. Structural dependencies in a system often induce strong correlations. Here, we order all pairs of strongly correlated metrics from the highest $R^2$ to the lowest. We then select pairs in this order until the size of the metrics selected reaches the number sought.

The two methods above make a coarse-grained use of the correlation information. However, further insight can be gained from the correlations. In principle, the subset selection methods should make use of as much of the correlation information as possible so that the majority of the system dynamics reflected in the correlations can be represented.

### Selection by Clustering

This method is predicated on the idea that groups of closely related metrics reflect some artifacts or specific dynamics in the system, and as such, we need to capture these groups of correlated metrics to better monitor the system. We can discover

| **Method** | **Cluster distance measure** |
|---|---|
| Single-linkage | $dist(C_i, C_j) = min_{v \in C_i, v' \in C_j} dist(v, v')$ |
| Complete-linkage | $dist(C_i, C_j) = max_{v \in C_i, v' \in C_j} dist(v, v')$ |
| Average-linkage | $dist(C_i, C_j) = \frac{1}{n_i n_j} \sum_{v \in C_i} \sum_{v' \in C_j} dist(v, v')$ |

Table 8.1: HAC clustering methods used

these groups automatically by applying statistical clustering to the metric data. Since groups of correlated metrics expose similar information, we can select representatives from each group or cluster. By doing so, we can retain a set of correlated metrics capturing the essence of the range of dynamics captured by the overall set.

This method presumes that when a fault affects some part of the system (*e.g.*, a group of interacting components), many correlations associated with that part are affected. As such, one or few of these correlations are likely to be sufficient to detect the fault. However, there is no guarantee that all correlations associated with that part are perturbed. The more representative metrics we select from a cluster, the less likely we are to miss anomalies, albeit incurring a higher overhead. Thus, we need to find an appropriate trade-off between the ability to detect faults and the monitoring overhead, especially given the performance penalty in a system that may only be partially operational.

To apply this selection method, we first identify stable metric correlations as described in Chapter 7. We then create a dissimilarity matrix $\mathbb{D}$ by setting the entry $d_{i,j} = 1 - R^2$ if the correlation between metrics $i$ and $j$ has passed validation. Otherwise, we set $d_{i,j} = 1.0$, which is the maximum value an entry in $\mathbb{D}$ can take.

We employ hierarchical agglomerative clustering (HAC) [47], which operates by considering each object as part of a separate cluster and merging the clusters into larger clusters until a stopping criteria is met or all objects become part of a single cluster. Several HAC methods exist. They differ in their definition of the distance between clusters. We consider three methods: single-linkage, complete-linkage, and unweighted-average-linkage. While in single-linkage the distance between two clusters is given by the distance between the nearest neighbours from the two clusters, in complete-linkage the cluster distance is a measure of the farthest neighbours from the clusters. Average-linkage provides a middle ground between the two methods by averaging all pairwise distances between members of the two clusters. Let $dist(v_1, v_2)$ be the distance between metrics $v_1$ and $v_2$, $C_i$ be a cluster, and $n_i$ be the number of metrics in $C_i$. The notion of distance used by the three methods is summarized in Table 8.1.

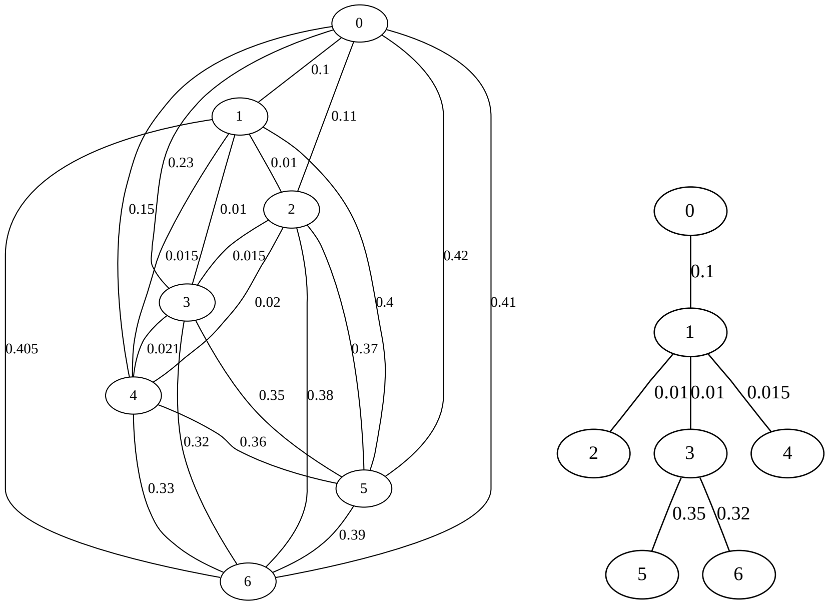
```
Algorithm 1:   Metrics subset selection using clustering
Input:  $\mathbb{D}$ (distance matrix), k (number of metrics to select), max_dist (maximum
        intra-cluster distance)
Output:  $S$
begin
    $S := \emptyset$;
    $C$ := cluster($\mathbb{D}$, max_dist);
    ;                                                  // obtain the set of metric clusters
    $L$ := sort $C$ by $|G|$, where $G \in C$;
    while $|S| < k$ do
        foreach $G$ in $L$ do
            $v_1$ := pick-random-metric($G$);
            $S := S \cup v_1$;
            remove($v_1$, $G$);
            if $|S| = k$ then
                break;
            pick $v_2$ randomly from $G$, where distance($v_1$,$v_2$) <= max_dist;
            $S := S \cup v_2$;
            remove($v_2$, $G$);
            if $|S| = k$ then
                break;
end
```

Because our goal is to find groups of closely related metrics, we use maximum intra-cluster distance as the stopping criterion in the clustering procedure. Our algorithm for metric selection using clustering is listed as Algorithm 1. The same algorithm is used with the three clustering methods described above.

The use of clustering for metric filtering presents several challenges. First, the right clustering method needs to be chosen. For example, the use of single-linkage often results in a small number of clusters, of which a few are relatively much larger; in contrast, complete-linkage tends to produce many clusters of small size. Second, it requires setting parameters (*e.g.*, maximum intra-cluster distance), which influence results heavily. Third, depending on parameters and the clustering technique used, it is possible for many metrics to end up alone (*i.e.*, in clusters of size one). Ensuring that such metrics are not ignored requires special handling. Finally, clusters only specify membership; within each cluster, the correlation information is ignored.

### Selection by Minimum Spanning Tree

We now propose a method which does not require setting any parameter, is equally or more efficient than traditional clustering algorithms, and retains more information than mere cluster membership. This technique originates from studies on correlations among financial equities in stock markets. In that context, correlation networks are formed by cross-correlating stock price returns. Researchers have used



(a) Correlation network (b) MST

Figure 8.3: An example of a correlation network and an MST derived from it

Minimum Spanning Trees (MST) as a tool to summarize the information content of such networks [15, 89]. The same research argues that the derived MSTs retain the most important information; as such, they can help create better stock portfolios.

An example of a correlation network and its MST is given in Figure 8.3. The tree is obtained by retaining the strongest correlations while ensuring that all nodes can be reached. It is not necessary for a correlation network to have a unique MST. When ties exist, different MSTs can be derived from the same network.

By mapping metrics to nodes and correlations to links, we obtain one or more correlation networks. Our method is to summarize the networks in the form of MSTs and then to select metrics from them. The nodes in an MST differ according to the number of their children. Each node with children represents a group of metrics that are closest to each other. For example, nodes 3, 5, and 6 in Figure 8.3 form such a group. To select a subset of metrics, we randomly pick metrics from the parent-children groups that exist in an MST. Algorithm 2 provides an overview of our MST-based selection method. When there exists more than one correlation network, we can create MSTs for each network and combine the resulting sets of parent-children groups for use in Algorithm 2.

Given an $n \times n$ adjacency matrix, Prim's algorithm [29] can be used to derive an MST with time complexity $O(n^2)$, which is generally more efficient than the

```
Algorithm 2:   Metrics Subset Selection using MST
Input:   $\mathbb{D}$ (distance matrix),k (number of metrics to select)
Output:  $S$
begin
    $S := \emptyset$;
    mst := compute-mst($\mathbb{D}$);
    $T$ := obtain set of (parent,{children}) tuples from mst;
    $L$ := sort $T$ by |{children}|;
    while $|S| < k$ do
        foreach $(p, C)$ in $L$ do
            $S := S \cup p$;
            if $|S| = k$ then
                break;
            r := pick-random-metric($C$);
            $S := S \cup r$;
            remove($r$, $C$);
            if $|S| = k$ then
                break;
end
```

clustering techniques considered here. In fact, one efficient way of finding single-linkage clusters is to obtain the MST first.

## 8.2 Minimal Monitoring

To continuously oversee the health of the system, we need a minimal level of monitoring. We require models to track the behaviour of the metrics collected at this level.

### 8.2.1 Using Metric Correlation Models

As our system model is an ensemble of metric correlation models, we can simply choose the subset of these models that relate to the selected metrics. These models are continuously applied to the collected data. When these models repeatedly detect outliers, we hypothesize that there exist faults in the system. We then adapt monitoring to obtain more information in order to confirm or refute the hypothesis. More specifically, we track correlations by feeding new metric observations to the associated regression models and checking for outliers (*i.e.*, values that fall outside the acceptable intervals). Each model's residuals are recorded in a sliding window of length $w$. If a model detects outliers in the majority of $w$ the entries, it is considered to have failed.

To detect errors in the system, we aggregate results from individual models thus:

$$F_t = \sum_{m \in M} S_t(m) \tag{8.1}$$

where $t$ is the time, $M$ is the set of correlation models associated with metrics in the minimal set, and $S_t(m)$ is the assessment of a model $m$ given by:

$$S_t(m) \quad = \quad \begin{cases} 1 & \text{if } \sum_{i=0}^{w-1} s_{t-i}(m) > \frac{w}{2} \text{ and} \\ & \sum_{i=0}^{t} s_i(m) < (1-p)t \\ 0 & \text{otherwise} \end{cases} \tag{8.2}$$

where $s_t(m) = 1$ if an outlier is detected at time $t$ by model $m$ and 0 otherwise. $p$ is a parameter, with value in the range $[0, 1)$, specified by system operators. A model is considered reliable only if a proportion $p$ of past observations of the relevant metrics are not outliers. A model fails at time $t$ if it reports outliers for the majority of the most recent $w$ samples. $F_t$ represents the total number of reliable models that fail at time $t$; if $F_t > F_{max}^{MM}$, the monitoring system reports an error at time $t$.

The reason to only account for reliable models is to prevent false alarms. Unreliable models may have escaped our tests during model validation. It is also possible that the underlying metric relationships have evolved.

### 8.2.2 Augmenting Minimal Monitoring with Threshold-Based Models

Certain faults do not perturb metric correlations. In fact, it is possible for faults to strengthen existing correlations, or even induce new ones. As a result, despite there being a fault, correlations may continue to hold while the absolute values of the metrics involved reach abnormal levels. One approach to circumventing this limitation is to combine correlation models with other models that are less prone to this problem. We consider the use of single-metric models, which capture a metric's behaviour without relying on any other metric.

There are many techniques for creating single-metric models. The simplest model is a fixed value range which determines whether the metric's behaviour is within the norm. We refer to such models as *threshold-based models* because they can be seen as indicator functions predicting whether conditions of interest exist. More elaborate single-metric models such as autoregressive models allow one to make predictions based on past values of the metric. As a proof of concept, we employ threshold-based models. Such models are commonly employed in practice. For example, many businesses enter into service-level agreements (SLA) with other businesses or customers to deliver service according to given performance or reliability standards, commonly known as service-level objectives (SLO). An SLO is in essence a threshold-based model which in addition to a target level also specifies a measurement period and a minimum fraction of observations required to satisfy the target.

Similarly, setting thresholds on key metrics, such as resource utilization levels, is the most common technique used by system operators to spot anomalous metric behaviour. Industry products such as IBM Tivoli Monitoring [57] and HP OpenView [52] readily allow system administrators to set thresholds on such metrics. Furthermore, for key metrics, system operators may be able to determine what constitutes acceptable threshold values based on their knowledge and experience. For example, it is has been reported that Internet service users are likely to be annoyed if a page takes more than four seconds to load [74]; a threshold can thus be set to reflect such a user preference.

Appropriate threshold values are generally difficult to determine. This is especially true for metrics that belong to components that are not directly visible from the system's interface, as setting the threshold requires good understanding of the internals of the system. In the absence of any domain knowledge, we can analyze historical data to determine appropriate threshold values. Since the focus of our work is not on devising new schemes to determine thresholds, we avail ourselves of schemes already used in practice; we use a percentile-based technique to set thresholds in our evaluation.

The purpose of using threshold-based models is to enable a better assessment of the system's health during minimal monitoring. Irrespective of the technique used to set thresholds, we consider a threshold-based model to have failed when it is consistently violated for $t$ consecutive sampling intervals. This mechanism allows us to make threshold-based models more robust to transient disturbance. If the target system is already bound by SLOs, violations of the underlying thresholds correspond to failures. When such a violation occurs, the failure is readily known and there is no need for additional evidence to corroborate the failure. However, there is a need to localize the cause of the failure, which can be accomplished by proceeding to detailed monitoring.

There are circumstances where violations of thresholds do not necessarily correspond to failures. This may be case when thresholds have been derived automatically from historical data, or when the thresholds were set by system operators without a formal basis. In these cases, when threshold-based models fail, we need to obtain corroborative evidence to validate the observed anomalies. In our approach such evidence is obtained by enabling detailed monitoring and analyzing the metric correlation models.

With the addition of threshold-based models, we have two types of models at our disposition during minimal monitoring. Our basic monitoring approach is to combine both types of models. This approach has the potential to improve fault coverage because of the complementary nature of the two types of models. During minimal monitoring, if any one of the threshold-based models or the correlation models persistently detects anomalies, we proceed to detailed monitoring.

## 8.3 Detailed Monitoring

As discussed earlier, the detailed monitoring level represents a more-thorough oversight of the system by tracking a much larger set of metrics than minimal monitoring. Detailed monitoring, being costlier, is only enabled when we need more information to corroborate anomalies detected during minimal monitoring and to localize the source of the anomalies observed. Fault localization is discussed in detail in Chapter 9. Here, we focus on the validation aspect of detailed monitoring.

During detailed monitoring metrics are tracked using the correlation models we learned *a priori*. As in minimal monitoring, a regression model is considered to have failed if it consistently reports outliers. More specifically, we compute $S_t(m)$, which is the assessment of a model $m$, as follows:

$$S_t(m) \quad = \quad \begin{cases} 1 & \text{if } \sum_{i=0}^{w-1} s_{t-i}(m) > \frac{w}{2} \\ 0 & \text{otherwise} \end{cases} \tag{8.3}$$

where $s_t(m) = 1$ if an outlier is detected at time $t$ by model $m$, and 0 otherwise.

Note that in Equation 8.3, we no longer take into account any notion of model reliability. The reason is that for most models checked during detailed monitoring, we have no prior data to estimate reliability. The metrics associated with the majority of the models are only collected when detailed monitoring is enabled. As such, we have no record of how often most the models have failed since the time of the last error detection. In contrast, during minimal monitoring, we have a fixed set of correlation models that are always checked. We can thus compute a model reliability score by tracking how often they fail.

Since we do not know how reliable a model is in detailed monitoring, we allow some models to fail without suspecting errors in the system. This allowance is captured by the $F_{max}^{DM}$ parameter, which is described further below.

Each metric correlation model only provides an assessment of the metrics it covers. We thus need to combine the results from the individual models to evaluate the global system health. There are several natural abstraction levels at which this aggregation can be effected. Three aggregation levels which match our abstraction of the target system are: global (all), metric, and component.

Given an abstraction level, we aggregate results from all the available models and detect errors in the system thus:

$$F_t = \sum_{m \in M} S_t(m) \tag{8.4}$$

where $t$ is the time, $F_t$ represents the number of models that fail at time $t$, $M$ is the set of correlation models available for analysis at the aggregation level considered, and $S_t(m)$ is given by Equation 8.3. If $F_t > F_{max}^{DM}$, the monitoring system reports an error at time $t$. In general, if model validation is thorough, we expect $F_{max}^{DM}$ to be set to a very small value.

- **Global Level:** The global abstraction level entails taking account of the results from all the correlation models applied at the detailed monitoring level. At this aggregation level, $M$ in Equation 8.4 is the set of all the models available during detailed monitoring. If $F_t > F_{max(g)}^{DM}$, an error is reported.
- **Metric Level:** A modeled metric is correlated to one or more other metrics. We can analyze results from correlation models at the metric level, whereby errors are detected when a significant number of models associated with any one metric fail.

  Faults generally cause specific dynamics in the system to be perturbed. Such perturbances tend to affect specific clusters of correlated metrics. Thus, the metric-level analysis can be more sensitive to faults. In comparison with the global level, analysis at the level of metrics allows the different dynamics captured by metric correlations to be assessed separately. Perturbance may appear negligible when considered at the global level, but it can be significant when viewed at the metric level.

To detect errors, for each metric $v$, we use Equation 8.4, where $M$ represents the set of correlation models associated with metric $v$. If $F_t > F^{DM}_{max(m)}$ holds for at least one metric, an error is reported.

- **Component Level:** When a component fails, its metrics are likely to behave anomalously, which in turn may cause the associated correlations to be perturbed. Because we assume that the metric-to-component mapping is known, we can analyze results from correlation models at the component level. For each component $c$, we use Equation 8.4, where $M$ represents the set of correlation models associated with component $c$. If $F_t > F^{DM}_{max(c)}$ holds for any component, an error is reported.

Note that if the metrics tracked during minimal monitoring are chosen using the automated selection methods described in Section 8.1.2, $F^{DM}_{max(g)}$ needs to be larger than $(F_t + U_t)$. $F_t$ is number of models which failed at time $t$ and $U_t$ represents the number of models found to be unreliable at time $t$, both of which are computed during minimal monitoring. This essentially implies that for detailed monitoring to support any error hypothesis generated during minimal monitoring, it should find more correlation perturbance than what was observed during minimal monitoring.

Our adaptive monitoring approach relies on minimal monitoring for triggering the error detection logic. Therefore, errors can be missed if they are not detected at the minimal monitoring level. In such cases, detailed monitoring will not be triggered even though it might have provided pertinent information. If errors are missed systematically, the parameters of minimal monitoring need to be fine-tuned, taking care of striking the right balance between the ability to detect errors and the occurrence of false alarms. Errors can also be missed when anomalies detected at the minimal monitoring level are not confirmed by the analysis at the detailed level. In such a case, detailed monitoring is unable to support any error hypothesis and cannot help locate sources of faults. If the anomalies observed do not correspond to known failures (*e.g.*, SLO violation), the anomalies detected with minimal monitoring are not reported to the system operators. Though not reported, the detected anomalies can be logged; this data can help post-mortem analysis and assist in configuring the monitoring system.

## 8.4 Experiments and Analysis

In this section we describe experiments in which we use an implementation of our adaptive monitoring approach. Our target is the Trade system described in Chapter 5. Metric correlation models are identified by the approach presented in Chapter 7 and the modeling technique used is Simple Linear Regression.

Our evaluation is based on two data sets. The first data set consists of metric data collected over a period of 36 hours, during which the system is not intentionally subjected to any fault. Half of this data is used to learn and validate the models needed for monitoring. We use the remaining half to check for false alarms. As reported in Chapter 7, we identify 5138 SLR models, which cover a total of 224 metrics.

Our second data set includes data collected from a set of 39 fault-injection experiments. Each experiment involves injecting a fault in a component of either the system software or the Trade application. The fault is injected while the system operates normally. A detailed description of the faults we inject is available in Chapter 5.

In our evaluation, results from the correlation models are kept in a sliding window of length six (*i.e.*, $w = 6$), which represents a delay of one minute. We set $p = 0.99$ (*i.e.*, we only consider those models to be reliable for which 99% of observations are not outliers). We set $F_{max}^{MM}$ to 1 (*i.e.*, an anomaly is detected if at least one model fails persistently).

### 8.4.1 Minimal Monitoring: Manual Selection

We first evaluate our adaptive monitoring system using the manual metric selection approach. During minimal monitoring, a fixed set of manually selected metrics is tracked using threshold-based models, metric correlation models, or both.

Leveraging our domain knowledge, we choose to track metrics related to the response time of and failures in components which are accessed directly by the end users. In our experiments, this translates into monitoring the number of requests to the different web pages of the application, the time taken to deliver those pages, and the number of failed requests.

In order to evaluate our approach with threshold-based models, we set thresholds based on past normal behaviour. Using historical data, the threshold for the upper bound on acceptable values for a metric is computed as follows:

| Model type | Num. false alarms | Num. faults detected |
|---|---|---|
| Threshold-based models | 48 | 30 |
| Correlation models | 0 | 7 |
| Combined | 48 | 37 |

Table 8.2: Minimal monitoring detection results

$$T_{max} = \text{percentile}([x_i], p)(1 + \text{markup}) \tag{8.5}$$

where $T_{max}$ is the maximum response time threshold, the function *percentile* computes the $p$-th percentile from the vector of observed response time values $[x_i]$, and markup represents an additional non-negative margin. If $T_t > T_{max}$ for $k$ consecutive samples, we report anomalies.

In our evaluation, we use Equation 8.5 to determine thresholds for response-time metrics. For failure metrics, we take the conservative view that any persistent failure is worth investigating. If any failure metric is non-zero for at least $k$ consecutive samples, we report anomalies. We do not track request counts using threshold-based models, as it is not semantically meaningful. We set $k$, the number of required repeated violations, to 3; in practice, however, $k$ should be set according to the needs and particulars of the system being monitored. Although we only employ static thresholds in this work, thresholds can also be dynamic. For instance, the percentile or any other aggregation function can be based on a sliding window of past values of a metric.

Table 8.2 summarizes the detection results for minimal monitoring using the manual metric-selection approach. In this set of experiments, thresholds for response-time metrics are set using $95^{th}$ percentile of learning data and a markup of 0.1. These results indicate that we can detect more faults with the threshold-based models, but we suffer from a higher level of false alarms. Metric correlation models, on the other hand, detect fewer faults, but do not suffer from any false alarms. The noteworthy result is that the two types of models complement each other, for together they detect 35 of the 39 faults. The correlation models are able to detect faults that are not reflected in the web page failure metrics and the faults that do not significantly impact response-time. On the other hand, thresholds allow us to readily detect those faults that are reflected in web page failure metrics and that cause large increases in response time.

Our results show that augmenting correlation models with threshold-based models improves system monitoring. The combination allows us to improve fault cov-

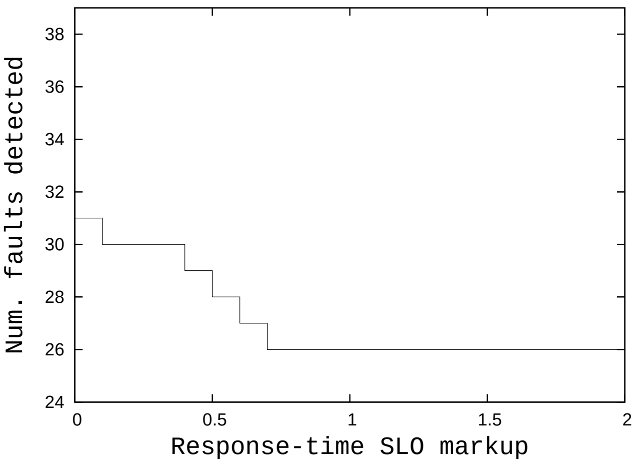


Figure 8.4: Effect of varying SLO markup on fault coverage

erage. There are two reasons that underlie this improvement. First, some faults may not cause correlations to break even though the metrics involved soar beyond acceptable levels. Such faults can be detected by threshold-based models. Second, some metrics are tracked more easily using threshold-based models; the same metrics may not be associated with any correlations. For instance, it is easier to define an upper limit for metrics which reflect the number of errors or failures in a system. With our approach, these metrics are not associated with any correlations. When identifying correlations, we discard all metrics that do not display any variance. Because correlation identification takes place while the system is in a healthy state, the error metrics mostly report nil. Third, certain metrics are more naturally modeled with correlations than thresholds. For instance, the request counts for the application web pages are tracked readily using metric correlation models; tracking these with thresholds is non-trivial because these vary with the workload.

In order to study the incidence of false alarms, we vary the markup used to compute the thresholds for web page response times. We do not vary thresholds on the number of web page failures, for any consistent web page failure is a cause for concern. The results shown in Figures 8.4 and 8.5 confirm the trade-off between sensitivity to faults and false alarms. Without any markup, we can detect 31 faults, but we would incur 135 false alarms. With a markup value that avoids false alarms, we only detect 26 faults.

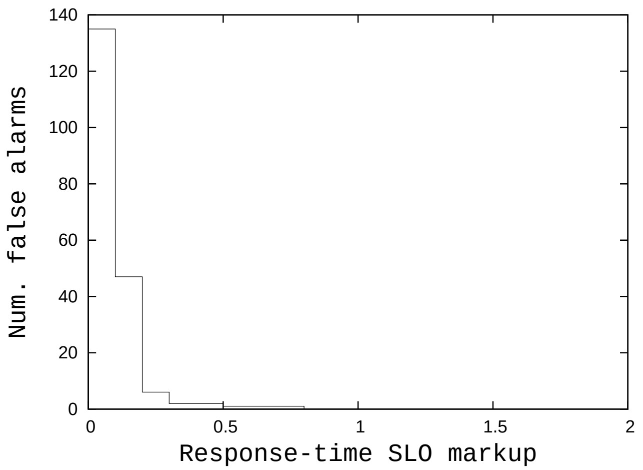


Figure 8.5: Effect of varying SLO markup on false alarms

### 8.4.2 Minimal Monitoring: Automated Selection

We now evaluate the alternative approach to selecting metrics for minimal monitoring. For clustering-based selection, we fix the maximum intra-cluster distance to $(1 - R^2_{min})$ (*i.e.*, 0.4). Since there is no unique MST when there are correlations of equal strength (*i.e.*, equally distant), we repeat each analysis ten times by randomizing the metric indices in the distance matrix $\mathbb{D}$. This allows ties to be processed differently in each repetition. This also allows different decisions to be made where a random choice is involved. The following results present the mean number of faults detected and the 95% confidence intervals for the mean when different percentages of available metrics are selected for monitoring. For instance, 0.1 denotes using 10% of the modeled metrics for monitoring.

In all our experiments, no false alarms were reported using the parameters described above. A determining factor in avoiding false alarms is the use of reliable models only. Remember that our error detection procedure only takes into account those models for which 99% of the past observations fall within the acceptance bounds of the models.

We now compare the fault coverage of the methods described in Section 8.1.2. Figures 8.6, 8.7, and 8.8 [1] indicate that selection by clustering generally produces much better fault coverage than selection by the strongest correlation. The figures suggest that choosing by the strongest correlations is not a good strategy; it performs worse than random selection. This is so because the selected metrics tend to reflect the same underlying system dynamic, thus limiting the coverage.

[1] Note that in some figures we have shifted some curves slightly to improve clarity

The clustering methods provide a small improvement over random selection, though single-linkage performs slightly better than the other two methods. In this set of experiments, the use of single-linkage clustering outputs one very large cluster and a few small clusters (of size 2 to 6). Selection in the large cluster is random. Still, we obtain better fault coverage because the small clusters are guaranteed to be covered by the clustering-based selection method; this is not the case for pure random selection.

Average-linkage and complete-linkage produce more clusters with a more-varied cluster size distribution. Yet, they do not provide a marked improvement over random selection. This is explained by the fact that the clusters contain strongly correlated metrics. This indicates that focusing on the strongest correlations, even when they belong to different clusters, is not necessarily effective. In addition, the two methods leave many metrics in singletons because they are associated with weaker correlations. Such metrics cannot be selected unless additional heuristics are used in the selection method.

Figure 8.9 shows that MST-based selection performs much better than naïve selection. Figure 8.10 compares MST-based selection with clustering-based selection. We first see that MST consistently improves fault coverage over single-linkage. While MST performs better than complete- and average-linkage overall, for very small fractions of the available metrics, the clustering methods have a slight advantage. When the choice is limited to very few metrics, the two clustering methods are likely to choose metrics from clusters that capture different system dynamics, which is important for fault coverage. However, as the restriction on the number of metrics is relaxed, these methods fail to cover weaker correlations.

Our results suggest that selection of metrics subset by MST generally provides better fault coverage than naïve selection and clustering-based selection. MST not only has better fault coverage than clustering, but also does not need require any parameter (*e.g.*, cut-off distance) and costs less in computation.

The principal factor underlying the better performance of MST-based selection is that it not only captures strong correlations but it also retains groups of metrics that are closest, albeit less strongly correlated. A metric $v$ can have several other metrics as children because no other metric has stronger correlation to the children of $v$ than $v$ itself; this grouping is formed even though $v$ may not be strongly correlated to the children. Clustering, on the other hand, is intended to capture groups of strongly correlated metrics; weaker correlations are only included as side-effects. For example, when using single-linkage, metrics are added to a cluster based

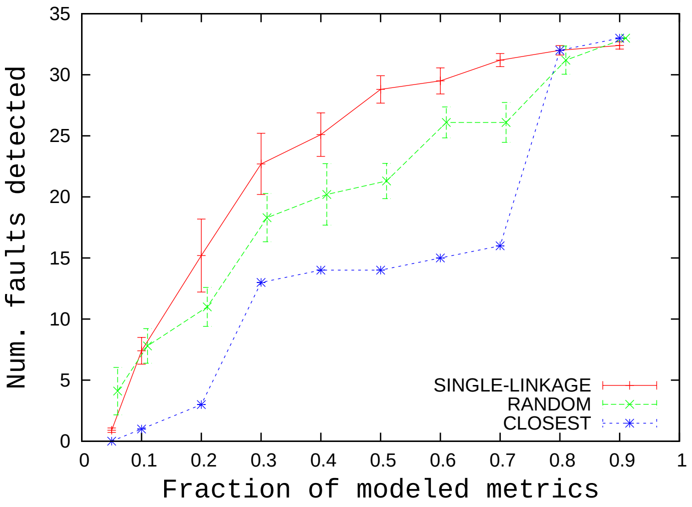


Figure 8.6: Single-linkage clustering *vs.* naïve selection

on their distance to the closest member of the cluster; this procedure, however, results in clusters having members that are not necessarily close. Despite the fact that with single-linkage we retain weaker correlations, it is less effective than MST-based selection. With single-linkage, some clusters are often large, making the selection of weaker correlations less likely because they are mixed with potentially many strong correlations.

### Improving Clustering-Based Selection

In our evaluation thus far the maximum intra-cluster distance is set to reflect what we consider to be strong correlations (*i.e.*, the cut-off is set to $(1 - R^2_{min})$). Several techniques exist to measure the quality of clustering. We can try to use such techniques to improve our clustering and hope that such improvement translates into better metric selection (*i.e.*, one that achieves better fault coverage). One popular technique to assess clustering quality is Silhouettes [75]. The Silhouette score $s(x)$ of an object $x$ is an indication of how good the cluster assignment for the object $x$ is. The Silhouette score is given by:

$$S(x) = \frac{b(x) - a(x)}{max(a(x), b(x))} \tag{8.6}$$

where $a(x)$ is the average dissimilarity of $x$ to all the other objects in its assigned cluster, $b(x)$ is average dissimilarity of $x$ to objects of the neighbouring cluster that is closest to it. The range of $S(x)$ is $[-1, 1]$, where a value close to 1 indicates a

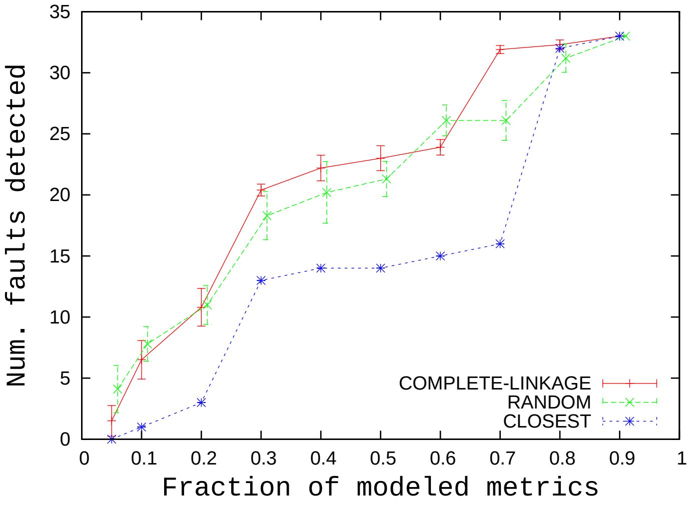


Figure 8.7: Complete-linkage clustering *vs.* naïve selection

good assignment (*i.e.*, $x$ is closer to the members of its own cluster than to those of the closest cluster). The average $s(x)$ of all objects in the data ($\bar{S}$) thus provides a measure of the overall quality of clustering.

In order to improve the clustering of our data, we searched for the cutoff distance that yields the highest $\bar{S}$ when using single-linkage clustering. We use a step size of 0.01 and repeat each analysis 10 times. We consistently obtain a cutoff distance of 0.06, corresponding to an average Silhouette score of 0.372. The fault coverage obtained from using this cutoff distance is depicted in Figure 8.11. We observe that the fault coverage is much worse than random selection. We repeated a similar analysis for both complete- and average-linkage and obtained similar fault coverage results. Two possible explanations for the results are: (1) there is no clear structure in the data, implying that clustering partitions the data artificially; (2) optimizing the aspect of clustering quality captured by the Silhouette score is not suitable for our purpose.

Despite the fact that the Silhouette score does not help improve fault coverage, our experiments suggest that an appropriate cutoff distance can help increase fault coverage significantly. For example, Figure 8.12 shows the effect of varying the cutoff value on fault coverage when using selection by single-linkage clustering with 30% of the available metrics. We can see that choosing a threshold value between 0.12 and 0.34 would increase fault coverage substantially. However, to make such a choice we would need to have access to fault data in advance, which is not practical. This result suggests that better fault coverage can be achieved by identifying the right properties of clustering which need to be optimized; the Silhouette score,

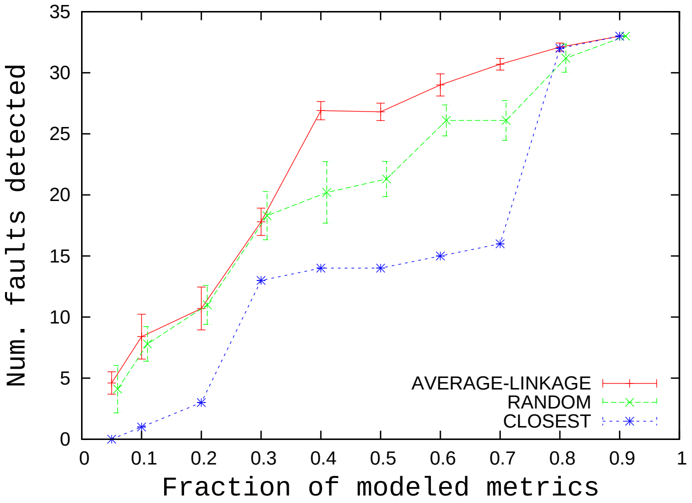


Figure 8.8: Average-linkage clustering *vs.* naïve selection

however, does not match this requirement. A deeper investigation of clustering quality in this context is left for future work.

Overall our results demonstrate that we can automatically select a subset of the available metrics that is effective in detecting errors in a system. Our results suggest that selection of the subset by MST generally provides better fault coverage than naïve selection and clustering-based selection. With MST-based selection, we can detect more than two-third of the faults by only tracking one-third of the metrics modeled with correlations.

### 8.4.3 Detailed Monitoring

In this section we evaluate the monitoring system's ability to detect errors when only detailed monitoring is used. During detailed monitoring, we only rely on correlation models to track the system's health. Remember that a total of 5138 SLR models are available to monitor the Trade system. We set $F^{DM}_{max}$ for all three aggregation levels to 5 ($i.e.$, $F^{DM}_{max(g)} = F^{DM}_{max(m)} = F^{DM}_{max(c)} = 5$), which corresponds to approximately 0.1% of the overall set of models. This value is small enough to provide a cushion against false alarms, detect any significant disturbance in the system, and potentially help in problem determination.

The detection results, presented in Table 8.4.3, indicate that the analysis of correlation models when aggregated at the metric and component levels is slightly more sensitive to the injected faults. This higher sensitivity reflects the ability to capture local disturbance using these aggregation levels. However, the higher

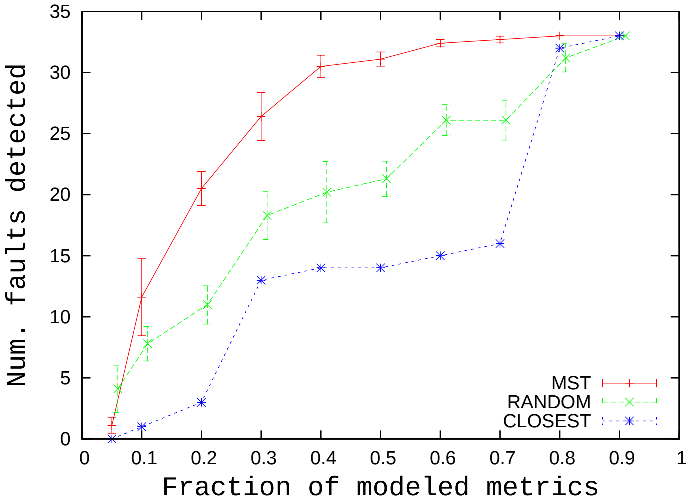


Figure 8.9: MST *vs.* naïve selection

sensitivity comes at the cost of more false alarms; we elaborate on these alarms further below.

| Aggregation level | Num. faults detected | Num. false alarms |
|---|---|---|
| Global | 32 | 0 |
| Metric | 36 | 97 |
| Component | 36 | 97 |

Table 8.3: Detailed monitoring detection results

Figure 8.13 shows the effect of varying $F_{max}^{DM}$ and aggregating the results from the correlation models at the three abstraction levels. We observe that analysis at the metric level is the most sensitive to faults, while the global level is the least sensitive. Results of the component-level aggregation lie between the two other levels.

The false alarms results are shown in Figure 8.14. These show that the global-level aggregation is the least vulnerable to false alarms. Results of the component-level aggregation closely follow those of the global level. In contrast, aggregation at the metric level suffers from the highest level of false alarms, only reaching a low level when $F_{max}^{DM} = 0.5$. One important reason underlying this high level of false alarms is the fact that a significant number of the modeled metrics have few correlations. Figure 8.15 depicts the cumulative distribution of the number

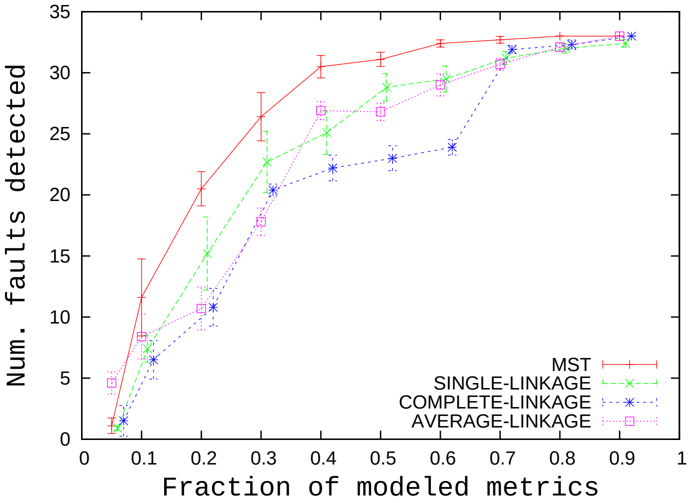


Figure 8.10: MST *vs.* Clustering-based Selection

of correlations per metric. We see that almost 20% of the modeled metrics are involved in one correlation only. As such, the $F^{DM}_{max}$ threshold is violated easily when even one metric exhibits anomalies.

From the above results, it may appear that the component aggregation level offers a good trade-off between sensitivity to faults and robustness against false alarms. However, further investigation of the false alarms reveals that they originate from 10 models only (from a total of 5138). As false alarms are investigated, the corresponding models will be discarded from the model ensemble, leaving the most reliable models and reducing eventual occurrences of false alarms. Therefore, a good strategy for detecting anomalies is to analyze the models at all three aggregation levels and set $F^{DM}_{max}$ to a small value such that the monitoring is sensitive to any significant disturbance. Aggregation at the metric and component levels ensures that the monitoring system is sensitive to local disturbance, while with global-level aggregation, the monitoring system can detect subtle disturbances whose effect is spread out in the system.

### 8.4.4 Adaptive Monitoring

Combining minimal and detailed monitoring provides a mechanism to validate error hypotheses and reduce false alarms. The hypotheses are generated in the minimal monitoring stage with less information than what is available during detailed monitoring. Also, the hypotheses generated during minimal monitoring may be based on models that are not as robust as the correlation models used during detailed

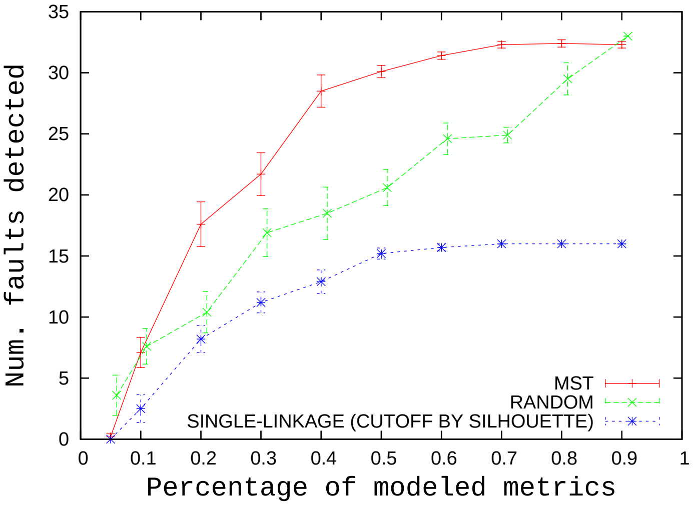


Figure 8.11: Single-linkage with cutoff distance selected by the Silhouette score

monitoring. For example, if we track response time metrics using the percentile-based thresholds as described in Section 8.2.2 with a markup of zero, the monitoring system will produce many false alarms. However, by leveraging detailed monitoring, none of these the threshold violations are reported, as the violations are not corroborated at the detailed monitoring level (*i.e.*, we do not observe a noticeable number of correlation models failing). Without the validation step, these violations would be reported to the system operators.

Table 8.4 summarizes the fault detection results of the adaptive monitoring approach. With the manually-selected metrics, the automatically determined thresholds allow detection of 30 faults, of which 25 are confirmed by detailed monitoring. In all, we can detect 37 of the 39 faults by combining threshold- and correlation-based models; 32 of these faults are confirmed by detailed monitoring. When the selection of the metrics for minimal monitoring is automated, we can detect 27 faults using the same number of metrics as we used with manual selection. All the 27 faults detected at the minimal level were corroborated by detailed monitoring. These results suggest that a completely automated approach to metric selection and tracking at the minimal monitoring level can produce results similar to those that rely on domain knowledge and human expertise.

We showed earlier that with detailed monitoring enabled on a continuous basis, we could detect 36 of the 39 faults we inject. However, this comes at the cost of greater performance overhead (12% as per our analysis in Chapter 6). In contrast, the cost incurred by our adaptive monitoring system is much lower. If we consider tracking the manually-selected metrics with combined threshold-based and corre-

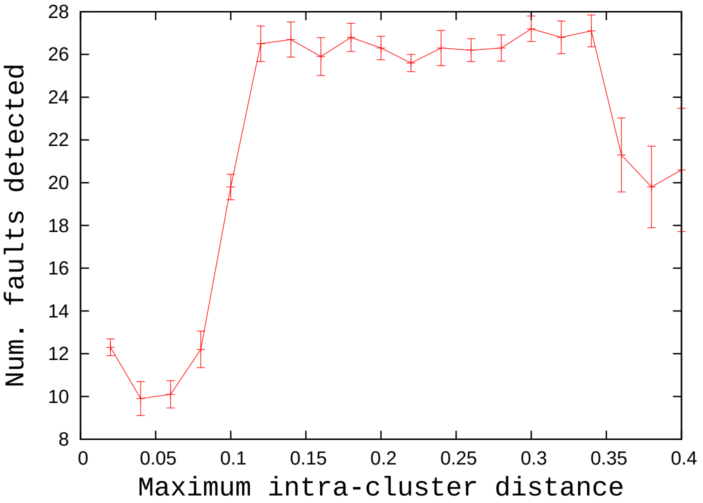


Figure 8.12: Effect of varying the maximum intra-cluster distance with single-linkage clustering

| Mechanism | Minimal Monitoring | Detailed Monitoring |
|---|---|---|
| Manual – Thresholds | 30 | 25 |
| Manual – Correlations | 7 | 7 |
| Automated – Correlations | 27 | 27 |

Table 8.4: Detection results with adaptive monitoring

lation models at the minimal monitoring level, we can detect 37 of the faults and experience 48 false alarms in a period of 18 hours. We thus have a false alarm rate of approximately three per hour. In our experiments we require a minimum of six samples (*i.e.*, one minute) for our analysis at the detailed monitoring level. Therefore, the average service time can be approximated by $\frac{57}{60}(3.88) + \frac{3}{60}(4.28)$. This represents only a 2.4% overhead. Even if we use two minutes worth of detailed monitoring for each false alarm, the performance overhead is less than 3%.

## 8.5 Adaptive Monitoring: Further Considerations

We next briefly discuss several issues pertaining to the implementation of an adaptive monitoring approach.

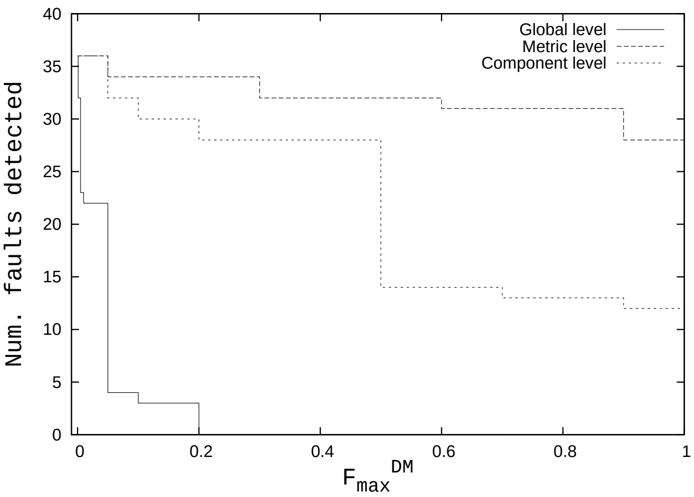


Figure 8.13: Effect of varying $F_{max}^{DM}$ on fault coverage

### 8.5.1 Combining Manual and Automated Metric Selection

In our evaluation, we have assumed that either manual or automated metric selection is used. Automated selection assumes the availability of no other information besides correlations. In practical monitoring scenarios, however, some metrics will have to be collected because either their collection is mandated by system operators or is recommended by software vendors. Both clustering- and MST-based metric selection involve choosing metrics from a group or cluster. One way to combine both approaches is to replace metrics that are selected automatically by equivalent metrics whose collection is required. Here, equivalence refers to metrics that are in the same group or cluster. This method does not guarantee that all manually-selected metrics will be substituted; we can add the metrics which could not be substituted to the minimal monitoring set. It is, however, important to ensure that the total cost of minimal monitoring remains within the desired overhead budget. If this is not the case, we can reduce the target number of metrics the automated method needs to select.

### 8.5.2 Using an Intermediate Monitoring Level

The adaptive monitoring approach of this chapter uses two levels of monitoring: minimal and detailed. When faults are suspected, we enable detailed monitoring directly. This raises a several issues, especially when the overhead of detailed monitoring is high.

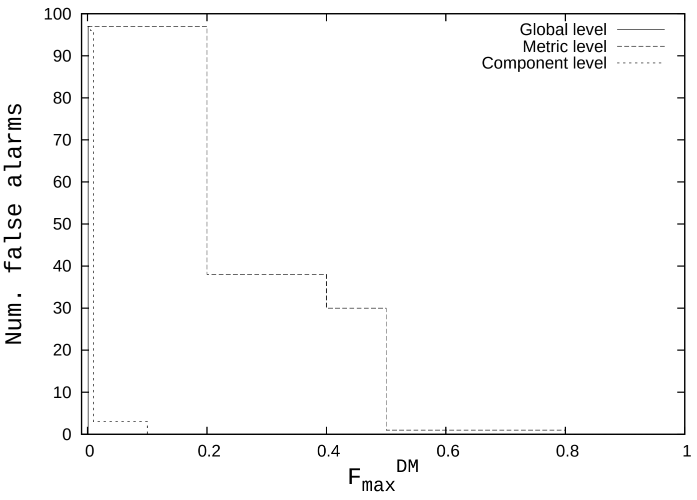


Figure 8.14: Effect of varying $F_{max}^{DM}$ on false alarms

- If minimal monitoring suffers from a high false-alarm rate, the overall cost of monitoring will become high, since each false alarms would increase the monitoring level.

- Certain anomalies detected during minimal monitoring via threshold-based models may reflect genuine errors, which the correlation models cannot detect. In such cases, triggering detailed monitoring, which only employs correlation models, is wasteful.

- When a system operates close to saturation, enabling detailed monitoring will make the system unstable. This instability arises from the reduced efficiency caused by the detailed monitoring and the system having accepted more work than it can handle.

To address these issues, we can introduce an intermediate monitoring level. The purpose is to see if the analysis of correlation models corroborates anomalies observed during minimal monitoring and thereby determine if detailed monitoring will be useful. At this new level, we probe a subset of the correlation models. If we do not detect any anomalies, then we save the cost of the unnecessary detailed monitoring, albeit at the risk of missing anomalies that detailed monitoring could have detected. Otherwise, we can proceed with the full-scale detailed monitoring. Because the overhead of the intermediate level lies between minimal and detailed, it allows the system to adjust gradually to the reduced efficiency caused by the more-expensive monitoring levels (*e.g.*, by accepting less work). This helps avoid instability when the system is already operating close to saturation.

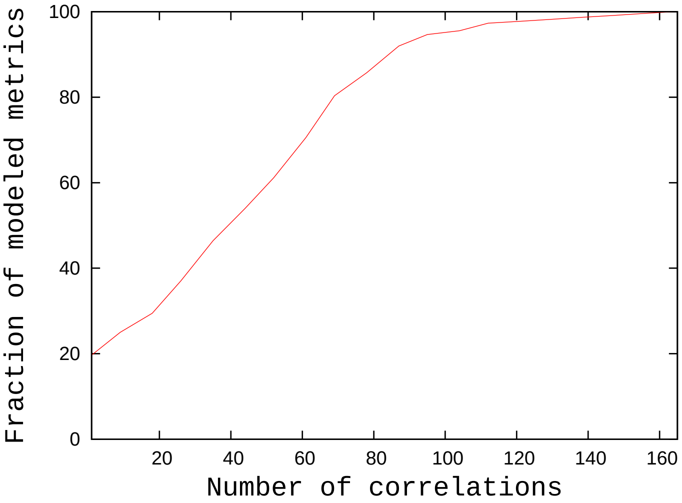


Figure 8.15: Metric correlation distribution

The metrics to be collected at the intermediate monitoring level can be selected using the methods described in Section 8.1.2. The choice will depend on an operator-defined performance overhead budget. To achieve overhead reduction, this budget needs to be smaller than that of detailed monitoring.

### 8.5.3 An Alternative Adaptive Monitoring Approach

In the course of this work, we considered an alternative approach to adaptive monitoring. It entails using an algorithm to discover dynamically which set of metrics it would be most pertinent to collect when anomalies are detected during minimal monitoring. Devising such an algorithm is difficult without knowledge of the system structure. However, we can devise an algorithm that makes use of the metric correlation information. When a metric in the minimal monitoring set behaves anomalously, this algorithm enables collection of those metrics correlated with the anomalous metric. For this algorithm to be effective, it is important to have representatives from each group of correlated metrics in the minimal monitoring set. If a cluster of correlated metrics does not have representatives in the minimal monitoring set, then the cluster would not be analyzed, as it is not reachable. The monitoring overhead can be reduced by employing a clustering technique that partitions the set into small clusters. As a result, anomalies are detected during minimal monitoring would only trigger the additional monitoring of small clusters of metrics.

This algorithm suffers from several shortcomings. First, it requires the use of clustering techniques. As discussed in Section 8.1.2, the use of clustering presents several challenges. The effectiveness of the algorithm depends heavily on the clustering method and the parameters used. Second, despite the use of representatives from all clusters of correlated metrics, we can fail to detect anomalies if the representative metrics are not affected; our automated metric selection methods also share this shortcoming.

### 8.5.4 Dealing with Slow Fault Resolution

Fault resolution can be time-consuming, especially when it involves interaction with software developers or vendors. As a result, a fault may continue to exist long after it has been identified. If we simply revert to minimal monitoring each time errors are confirmed and diagnosis performed, then the monitoring system will repeatedly enable detailed monitoring. This is inefficient, as the overhead of detailed monitoring will be incurred unnecessarily, slowing down the part of the system that is still healthy. In addition, the same information will be reported repeatedly, potentially wasting system operators' time.

Two approaches to dealing with this problem include caching of monitoring results and partial model deactivation. In the first approach, we keep a cache of the detection results during minimal monitoring. If a newly detected anomaly involves metrics which were reported and confirmed in the recent past, we can skip detailed monitoring under the presumption that it is a re-occurrence. The difficulties with this approach include defining what represents recent past and dealing with multiple, independent faults that occur within the same time frame.

The second approach is to deactivate temporarily those models that have failed because of a fault that is yet to be fixed. With our modeling approach, this implies removing the affected correlation models from our ensemble of models as well as any affected threshold-based model. The shortcoming of this approach is that it requires manual intervention; when the fault is resolved, the monitoring system needs to be informed so that the deactivated models can be reinstated. Resolving some faults may also require updating the application or the system software, which may require that our system model be re-learned. In that case, manual intervention would be difficult to avoid.

### 8.5.5 Keeping Metric Correlation Models Up-to-date

Software systems are often subject to change; software updates and patches, for example, are common occurrences. Likewise, user behaviour can also change. These changes can affect metric correlations, causing the model parameters to change or even inducing new correlations. Therefore, the correlation models need to be checked when such changes occur. If no mechanisms exist to keep abreast of such changes, then the correlations need to be re-evaluated from time to time to ensure validity.

While the system is healthy, which we expect to be the case most of the time, only the metrics collected during minimal monitoring are available for analysis. We therefore need to enable the collection of those other metrics. Two options exist in this regard: we can enable collection of the metrics we have already modeled, or we can enable collection of metrics regardless of whether they were modeled before. The latter option has the advantage of allowing new correlations to be identified. Since enabling the collection of all metrics at runtime would incur high cost, we need to limit the collection to subsets such that the overhead does not exceed a specified budget. Over time, we can cover the space of metric pairs, while the system operates normally. We discuss this idea further as an opportunity for future research in Chapter 10.

## 8.6 Summary

In this chapter we describe our approach to adaptive monitoring, which is enabled by our system model comprising an ensemble of metric correlation models. Our approach involves pre-specifying a fixed number of monitoring levels, corresponding to the monitoring of different subsets of the available metrics. Adaptation takes the form of transitions between the monitoring levels. We present the approach using two levels of monitoring, namely minimal and detailed. We propose automatic methods to select metrics to monitor at the minimal level. We also propose the use of threshold-based models to overcome some limitations of metric correlation models. We present techniques to combine metric-level results in order to gauge the overall health of the system.

By means of fault injection experiments, we evaluate the effectiveness of our monitoring approach in detecting errors and in avoiding false alarms. More specifically, we study the performance of minimal monitoring, detailed monitoring, and their combination in an adaptive monitoring system. We show that correlation-based monitoring effectively detects the vast majority of the faults we inject. We further show that, even with the availability of domain knowledge, the use of metric correlations improves error detection during minimal monitoring. More importantly, we show that an adaptive monitoring system can detect a significant portion of the faults at a fraction of the cost of detailed monitoring. We demonstrate how a completely automated, adaptive monitoring system can detect 70% of the faults using 30% of the modeled metrics, without relying on any domain knowledge or human expertise. Finally, our cost analysis shows that the cost of adaptive monitoring is slightly higher than that of minimal monitoring and much lower than detailed monitoring. Adaptive monitoring allows us to get the benefits of detailed monitoring without its cost.

One important function of adaptive monitoring is to provide detailed data in the event of faults to enable diagnosis. The next chapter describes our approach to localizing faults in the system.

# Chapter 9

# Diagnosis

The failure of software systems can have damaging consequences for organizations, including preventing normal operation and impacting goodwill. Despite the best software engineering and system management practices, errors and failures still occur. To limit the impact of these errors and failures, it is crucial to identify their causes quickly and take remedial action. However, the size and complexity of modern software systems make these tasks difficult, even for skilled and knowledgeable system operators. The purpose of this thesis is to reduce the involvement of human operators in system monitoring and problem determination tasks. In this chapter, we tackle the problem of automatically localizing faults. The problem of automatic recovery from errors and failures is a vast research area in its own right and is not investigated in this work.

Two factors make the task of fault localization time-consuming. First, software systems typically comprise many interdependent components and layers, which makes finding the source of errors difficult. Second, software system can expose a sea of complex data; finding relevant information manually in this data is a difficult task. We address the fault localization challenge by augmenting the monitoring system with automated diagnosis capabilities. As with error detection, we do not assume the availability of any information about the internals of the system, nor do we assume prior knowledge of faults. We devise diagnosis algorithms that use metric data and rely on a metric-based system model.

Intuitively, we expect diagnosis algorithms to work better when more data about the system health is available. However, collecting such extensive monitoring data incurs high performance overhead. However, one of our requirements for monitoring is that the performance overhead be low. This problem represents our motivation for devising an adaptive monitoring solution. In previous chapters we described an approach to system modeling that is suited to adaptation, and a monitoring approach that provides detailed data only when required. Detailed monitoring is not only needed to validate error hypotheses but also to enable fault localization. As such, the monitoring system performs diagnosis when the most metrics are collected. Specifically, our diagnosis algorithms are executed when two conditions are satisfied. First, errors or failures are suspected during minimal monitoring. Second, a global analysis of regression models during detailed monitoring indicates that the system is not in a healthy state.

As described in Chapter 7, our system model is an ensemble of metric correlation models. Alhough it is known that "correlation does not imply causation," correlation often provides useful insights about phenomena that underlie observations. The insight behind our diagnosis approach is that correlations capture regularity in a system's behaviour, which is imposed by the system's structure, which is a signature of the system in the healthy state; in the event of faults, correlations associated with the faulty components suffer from the most perturbance.

In this chapter we elaborate on how to leverage our system model to diagnose faulty components. Given our assumptions, there are two factors that make it difficult to pinpoint specific components as faulty. First, many component dependencies exist within a software system; it is difficult, with the correlation information alone, to disambiguate a faulty component from those that depend on it. Second, when a given stable correlation is perturbed, we cannot readily determine which one of the associated metrics is at fault. Without knowledge of component dependencies, we cannot determine the direction of the causal relationship between the metrics, if one exists. Therefore, rather than singling out a specific component, our diagnosis algorithms provide a list of components deemed to be faulty. We assign to each reported component an anomaly score reflecting the degree to which it is believed to be faulty. Such localization allows operators to quickly pin down the faults, albeit with some manual effort. It is our thesis that, in the absence of any other information besides what is available from the analysis of correlations, the operators can save much time by inspecting the components in the order reported.

The diagnosis approach we discuss here is not the only way to identify faulty components. Other sources of information (*e.g.*, log records) can contain valuable information to identify faults. The output of our diagnosis algorithms (*i.e.*, the list of components and their scores) can be combined with the output of other, ideally independent, algorithms to improve diagnosis results.

## 9.1 Analyzing Regression Models

Figure 9.1 depicts our view of the target system; each model is associated with metrics and metrics belong to components. This view suggests an intuitive, bottom-up, approach assessing the health of individual components. We can check whether a metric is anomalous by analyzing results from the models associated with it, and then reason about the components by analyzing the metrics or the models associated with them. When a component fails or experiences errors, its metrics likely behave anomalously, reflecting the change of behaviour or performance; the perturbed metrics may cause the associated models to fail. Using information about the failed models and the observations for which they fail, it is possible to quantify the extent of the perturbance.

Our overall approach is depicted in Figure 9.2. When detailed monitoring is enabled, we retain results of the correlation models in sliding windows. From these, we compute the model-level scores and perform further aggregation at the metric or component level. We first evaluate the correlation models individually, assigning each an anomaly score. We then assign anomaly scores at the level of metrics or components by aggregating model-level scores. Finally, we rank, short-list, and report the top anomalous components to system operators.

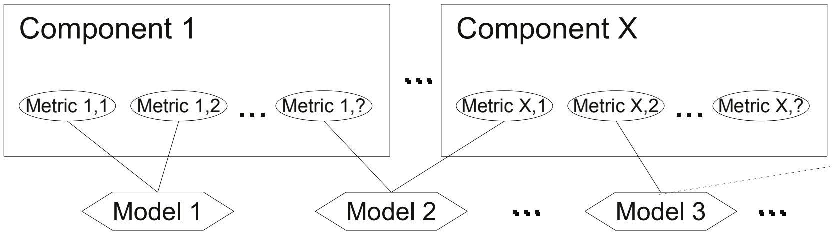


Figure 9.1: Relationship between components, metrics, and models

Each regression model provides several pieces of information which we can use to characterize the observed anomalies. First, the strength of the correlation between two metrics is known from when a correlation is identified. Second, during monitoring, we can check whether a model has detected outliers, and we can estimate the degree to which an observation is an outlier. Outlier degree is a function of the prediction error for a particular observation (*i.e.*, the difference between observed value and the predicted value for a target variable). We can use these indicators to compute an anomaly score for each model. There are many ways in which the

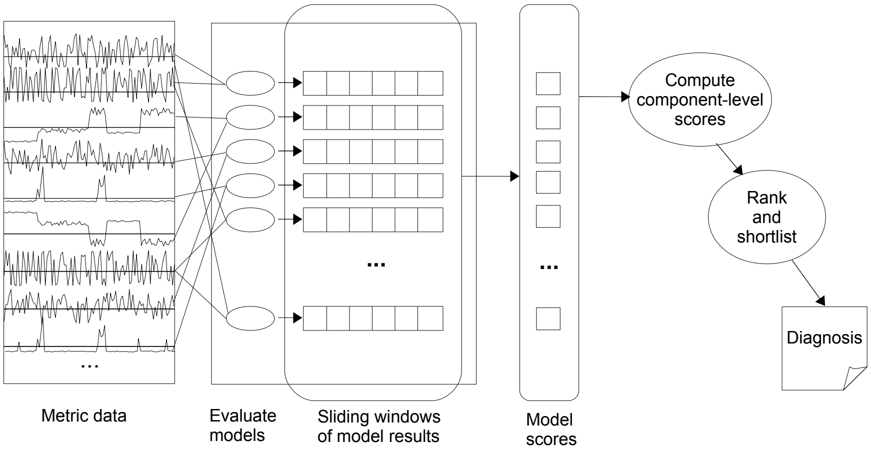


Figure 9.2: Approach to diagnosis

available indicators can be combined to compute anomaly scores; below, we present some basic approaches together with the underlying intuition.

## 9.2 Model-Level Anomaly Scores

Let $f(model, data)$ be the degree to which a model fits the learning data and $d$ be the outlier degree. Both $f(.)$ and $d$ are assume to have the range $[0, 1)$. In our work $f(.) = R^2$, an indication of the correlation strength, and $d$ is the studentized residual normalized by the maximum value of $d$ observed from the models considered in a given sample.

Four alternative anomaly score definitions for a model and a given sample are as follows:

$$
\begin{aligned}
S_1 &= \begin{cases} 1 & \text{if observation is an outlier} \\ 0 & \text{if observation is normal} \end{cases} \\
S_2 &= \begin{cases} f(.) & \text{if observation is an outlier} \\ 0 & \text{if observation is normal} \end{cases} \\
S_3 &= \begin{cases} d & \text{if observation is an outlier} \\ 0 & \text{if observation is normal} \end{cases} \\
S_4 &= \begin{cases} g(f(.), d) & \text{if observation is an outlier} \\ 0 & \text{if observation is normal} \end{cases}
\end{aligned}
$$

$S_1$ is the most basic score, whereby a model that reports outliers is assigned a score of one, and the correlation strength and the outlier degree are ignored. With $S_2$, the score of a model is the degree to which a model fits the learning data. The intuition behind this score is that the more correlated a pair of metrics is, the more anomalous it is when it fails to hold. The $S_3$ score is the normalized outlier degree. The more an observation fails to fit the model, the larger this anomaly score is. Finally, $S_4$ is a compound score based on both the normalized outlier degree and the goodness of fit. We use $g(f(.), d) = R^2 \times d$

As described in Chapter 8, to improve robustness, we require a model to consistently report outliers before it is taken to have failed. More specifically, we use equation 9.1 to determine if a model has failed.

$$
S_t(m) = \begin{cases} 1 & \text{if } \sum_{i=0}^{k-1} s_{t-i}(m) > \frac{w}{2} \\ 0 & \text{otherwise} \end{cases} \tag{9.1}
$$

where $s_t(m) = 1$ if an outlier is detected at time $t$ by a model $m$ and 0 otherwise and $w$ is the window length. The value of $d$ varies from sample to sample. Since a model's assessment is based on a sliding window, we set $d$ to the maximum value found in the window.

It is difficult for system operators to work directly with the set of models that report outliers, as their number can be large and many of the metrics these models cover can be shared. We therefore need means to combine model-level scores at the level of metrics or components.

## 9.3 Metric-Level Anomaly Scores

A plausible hypothesis we can make regarding a component that fails or has errors is that it will display the most anomalous behaviour. We gauge a component's behaviour through its metrics. Two intuitive definitions of what "most anomalous" metric means are:

- It is a metric that is associated with the most perturbed model. A model's level of anomaly can be quantified using any of the factors we discussed earlier.
- It is a metric whose associated metrics together display the highest degree of perturbance.

To formalize these notions, let $\mathbb{M}(v)$ be set of models associated with metric or variable $v$ and $S(m)$ be the anomaly score of a model $m$; $S(m)$ can be any of the scores described in Section 9.2. In order to find the most anomalous metric, we need to combine the model-level anomaly scores at the metric level and then choose the one with the largest score. Two ways to compute metric-level scores that capture the intuition behind our definitions are as follows:

1. **Max-score:** We assign each metric the highest anomaly score from the models to which it is associated. This global score is given by:

$$G(v) = \max_{m \in \mathbb{M}(v)} S(m) \tag{9.2}$$

2. **Ratio-score:** We sum the individual model scores such that contributions from all the associated models are included. Further, to put all metrics on equal standing, we normalize the sum by the maximum score which can result from the summation. This global score given by:

$$G(v) \quad = \quad \frac{\sum_{m \in \mathbb{M}(v)} S(m)}{|\mathbb{M}(v)| \times S_{max}} \tag{9.3}$$

## 9.4 Component-Level Anomaly Scores

We can also compute anomaly scores for system components. To this end, we consider three methods that make use of metric-level scores, model-level scores, or the number of anomalous metrics. Figure 9.3 depicts the methods with the information they require.

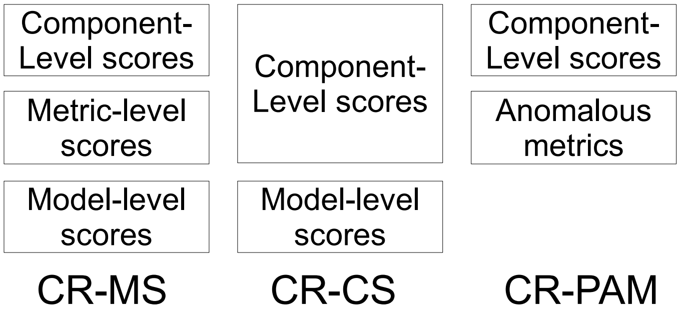


Figure 9.3: Types of component-level scores

1. **Component Ranking Based on Metric Scores (CR-MS):** The first method consist of assigning anomaly scores at the level of metrics as described in Section 9.3 and then extracting the component information from the reported metrics. In particular, we score the metrics, rank them, and instead of reporting metrics, we report the corresponding components. Several metrics belonging to the same component may be anomalous and thus be ranked; we only consider the rank of the first metric of each component. As a result, the component-rank may be much lower than the metric-rank of the first metric pertaining to that component, since several metrics that are ranked higher may belong to the same component.

2. **Component Ranking Based on Component Scores (CR-CS):** The second method is to compute the scoring functions described in Section 9.3 directly at the level of components. To this effect, we replace the metric $v$ by the component $c$ in Equations 9.2 and 9.3. We would thus use $\mathbb{M}(c)$, which is the set of models associated with metrics of component $c$, and the scores $S$ are aggregated per component (*i.e.*, $m \in \mathbb{M}(c)$). We can use the resulting anomaly scores directly to rank and short-list the most likely faulty components.

3. **Component Ranking Based on the Proportion of Anomalous Metrics (CR-PAM):** The third method involves computing, for each component, the ratio of metrics reported to be anomalous. A metric is considered to be anomalous if any model associated with it fails. Let $\mathbb{V}(c)$ be the set of modeled metrics pertaining to component $c$ and $\mathbb{O}(c)$ be the subset of $\mathbb{V}(c)$ (*i.e.*, $\mathbb{O}(c) \subseteq \mathbb{V}(c)$) which is considered anomalous. The component score is given

by:

$$G_3 = \frac{|\mathbb{O}(c)|}{|\mathbb{V}(c)|} \tag{9.4}$$

## 9.5 Reporting Diagnosis Information

The metrics or components that do not conform with the expected behaviour will have non-zero anomaly scores. We need to make two choices regarding what diagnosis information to present to system operators. First, we need to decide whether to report a ranked list of components or a ranked list of metrics. Second, we have to choose whether to report the top $k$ items or report all items with non-zero score to system operators.

With the anomaly scores described earlier, we have the ability to process and report either the most anomalous metrics or the most anomalous components. Reporting metrics raises two issues: First, there may be a large number of metrics that exhibit anomalous behaviour. Thus. it may not be easy to make sense of this list, especially if metrics of the same components are ranked far apart. Second, by limiting the list to a fixed size $k$, we may lose valuable information.

By reporting components instead of metrics, we can address the above issues to some extent. There are typically far fewer components than metrics in a system, so reporting components is less likely to confuse the system operators. Further, we can use more of the results of the analysis of metrics by combining that information at the level of components. Because of these advantages, in our evaluation we focus on the component-level diagnosis.

While we opt to report component-level diagnosis, we should point out that in practice diagnosis at both levels can prove valuable to system operators. Metrics carry extra information which may be lost if only components are reported. This information is often valuable in determining the nature of the errors or failures experienced. For example, performance-related problems tend to have a greater impact on timing metrics. Likewise, disturbance of correlations involving activity metrics often indicates anomalous changes in execution flow. An additional consideration is that metrics carry fine-grained details. For example, a metric may relate to a specific function of a component, which may assist operators in isolating the cause of anomalies quickly.

In our evaluation, we choose to report the top $k$ components based on the assigned anomaly scores. The parameter $k$ gives system operators some control over the use the diagnosis system. The chosen value of $k$ will depend on their confidence in the diagnosis system and the amount of time they are willing to spend investigating its output. The smaller the value of $k$, the faster the system operators can investigate the reported components. In the event that the actual faulty component or any of its metrics is not short-listed, the amount of time the operators will waste is limited. However, with larger values of $k$, the operators will lose much valuable time when the relevant item is not reported.

The alternative to presenting the top $k$ diagnosis is to report all anomalous metrics or components. While in this thesis we assume no knowledge of the system's internals, in practice system operators tend to have some such knowledge. If this is the case, the operators can find semantic linkages among subsets of the reported items, in particular when metrics are reported. For example, if an operator finds some database metrics reported together with metrics of the JDBC subsystem in an application server, he can readily suspect a fault related to data retrieval. If we restrict the list of reported metrics to a small length $k$, such analysis becomes more difficult.

An additional advantage of reporting the complete diagnosis results is that it allows system operators to get a sense of the extent to which system components are affected by faults. The reported anomaly scores give an indication of the degree to which the components are impacted.

## 9.6 Experiments and Analysis

We now present an evaluation of our diagnosis approach based on our 39 fault-injection experiments using the Trade system. For a description of the faults, the reader is referred to Chapter 5. In this chapter, we only evaluate diagnosis for the cases that can be detected by our monitoring system with the parameters described in Chapter 8. Our system model comprises metric correlations modeled with SLR. For error detection, we combine results from the global, metric, and component-level analyses; in our experiments, $F_{max}^{DM}$ is set to 0.1% for all three aggregation levels. Table 9.1 provides a summary of the results from these experiments.

Our approach to evaluating diagnosis accuracy and comparing different methods relies on the component ranks. A perfect result is for a faulty component to be ranked first. To evaluate the overall results, we use the cumulative distribution of faults with respect to the ranks of the true faulty components. In the figures

| | |
|---|---|
| Number of modeled metrics | 224 |
| Number of modeled components | 35 |
| Number of faults detected by detailed monitoring | 36 (3 missed) |

Table 9.1: Results from the monitoring of the Trade system using SLR models

presented below (*e.g.*, Figure 9.4 [1]), a point $(x, y)$ denotes the number of faults ($y$) for which the computed rank is $x$ or less. In essence, the more faults we can diagnose with a given maximum rank, the better the results are. We should point out that, in comparing two diagnosis algorithms, it may be preferable to consider differences for smaller rank values to be relatively more important. Smaller rank values imply that system operators can identify the faulty component faster. However, if there are major differences at the larger rank values, then more care is needed in interpreting the results.

Figures 9.4 and 9.5 shows the results of applying the Max-score (Equation 9.2) at the metric and component-level respectively. Remember that when the scoring function is applied at the metric level, we extract the component ranking from the results. The results show that for almost 2/3 of the cases where errors are detected, we can rank the faulty component within the top 15. This is much better than random ranking; if we order the 35 modeled components randomly, we have approximately 50% chance of ranking the faulty component within the top 17. More importantly, we can short-list the faulty component within the top 5 components in 15 of the 36 cases. These results provide evidence that valuable information can be obtained from the analysis of metric correlations.

Besides the overall results, we would like to know which model-level scoring function is the best and what aggregation level is the most adequate. In Figures 9.4 and 9.5, we observe that using $d$ for the model-level score produces the most accurate ranking. When using $d$ as the model-level score, there is no significant difference in the results if aggregation is performed either at the metric- or component-level (see Figure 9.8).

Figures 9.6 and 9.7 summarize the ranking results obtained by using the Ratio-score (*i.e.*, Equation 9.3). In both cases, we obtain more accurate results with the raw mode-level score (*i.e.*, 0 or 1) or $R^2$, although the advantage these scores enjoy is less clear when we directly aggregate the scores at the component-level.

Figure 9.8 compares the best result from the different algorithms discussed

[1] To improve the readability of the figures, we have joined the fault count results using lines; as such, line segments between successive ranks have no meaning

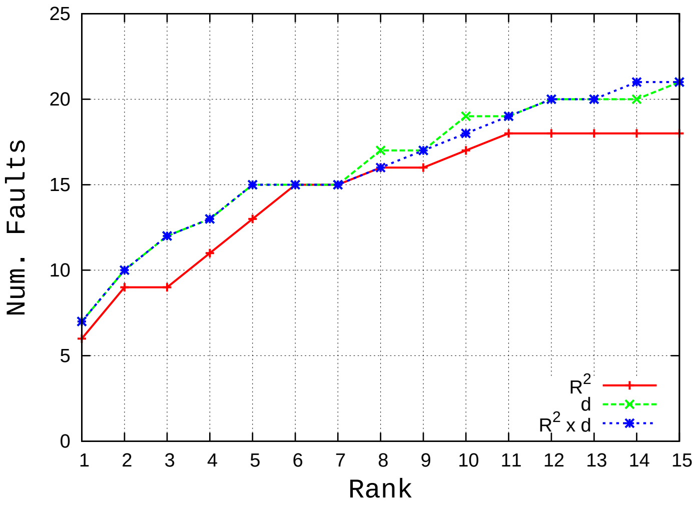


Figure 9.4: Diagnosis with Max-score and CR-MS

above. It also includes results of ranking components based on the ratio of anomalous metrics (*i.e.*, CR-PAM). These results show that computing the component-level score as a ratio of anomalous metrics is less accurate than alternatives based on aggregating model-level scores. We also find that the most accurate results are obtained with the following combination:

1. Using $d$ as the model-level anomaly score.
2. Using Max-score given by Equation 9.2 to aggregate model-level scores.
3. Aggregating anomaly scores directly at the level of metrics (CR-MS) or components (CR-CS).

While Max-score and Ratio-score are able to synthesize useful diagnosis information, we should point out that both have shortcomings. The main issue with the Ratio-score is that it ignores the number of correlations associated with a metric. For instance, a metric may be correlated with two other metrics; if the correlation with one of the metrics is perturbed, the score is 0.5. Another metric may be correlated with 100 other metrics; if 50 of the correlations experience perturbance, the score is still 0.5. However, the score in the first case is more sensitive in that it can vary greatly with small changes in the number of broken correlations.

However, metrics that have many correlations do not necessarily have more stable Ratio-score values. The more correlations a metric has, the more likely it is for many of these correlations to be accidental. When faults occur, many

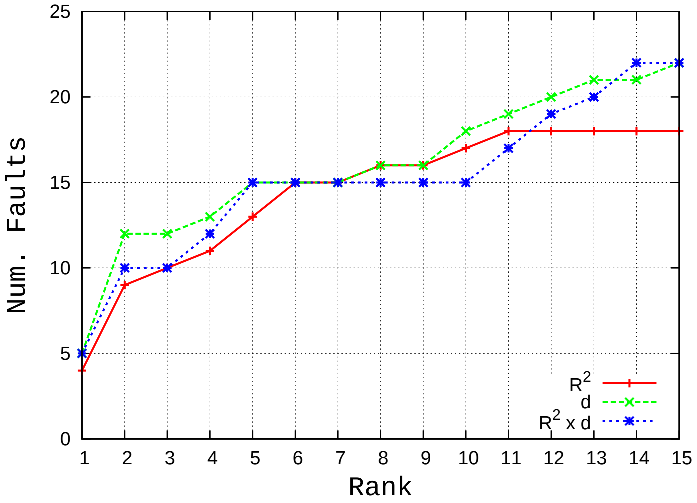


Figure 9.5: Diagnosis with Max-score and CR-CS

such correlations may break even though the metric is not anomalous. Consider, for example, a simple system with the components shown in Figure 9.9. Assume that a fraction of requests to A require execution of a method in B and another fraction require execution of a method in C. The metrics $m_i$ are activity counters, all of which are correlated with one another. The correlations involving $\{m_1, m_2\}$, $\{m_1, m_4\}$, $\{m_3, m_2\}$, and $\{m_3, m_4\}$ are incidental. If a fault causes C to fail, all these incidental correlations will break as shown in Figure 9.10. In this instance, components A and B have scores as large as that of C.

The particular example shown in Figure 9.9 and 9.10 also sheds some light on inaccuracies that arise because of dependencies in the system. Components that depend on faulty components may appear equally or even more anomalous than the true faulty component. In the figure, we see that $m2$ and $m4$ belonging to components A and C respectively have an equal number of broken correlations. In fact, it is possible for $m2$ to have more broken correlations if some correlations associated with $m4$ were not retained (*e.g.*, because of numerical imprecision or measurement errors) in the model identification phase.

The Max-score, on the other hand, eventually corresponds to the anomaly score of a single model. As such, it makes little use of the information provided by the other failed models. It is possible for correlations of components that depend on the faulty component to display more perturbance than correlations of the faulty component. In the example depicted in Figure 9.10, nothing prevents a broken correlation associated with component A or B to have the highest anomaly score. In fact, this may occur for reasons as simple as numerical imprecision when computing

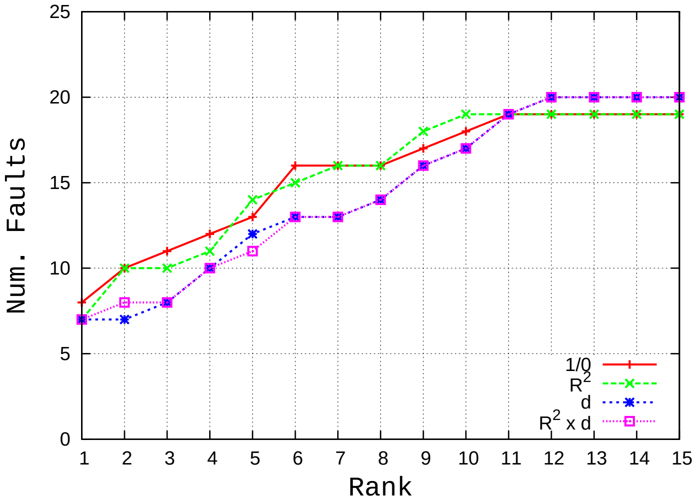


Figure 9.6: Diagnosis with Ratio-score and CR-MS

the outlier detection statistic, correlation strength, or the anomaly scores.

Despite these shortcomings, our results show that both Ratio-score and Max-score produce good diagnosis results, with Max-score being the better alternative. This is in line with the intuition that a faulty component is the one that displays the most anomalous behaviour. Nevertheless, further research is needed to see how we can overcome the limitations of correlation-based diagnosis. We describe some of our work to address this problem in Chapter 10 and other ideas in Chapter 11.

### 9.6.1 Nature of Faults and Diagnosis Accuracy

One key factor that determines diagnosis accuracy is whether faults perturb correlations associated with the faulty component. If pertinent metrics affected by a fault are not modeled, the diagnosis accuracy suffers. Likewise, if a fault causes the faulty component's metrics to become anomalous but does not perturb the metrics' correlations, the diagnosis accuracy suffers. To study such phenomena, we group our faults in two categories according to their expected effects and evaluate the diagnosis results accordingly. *Performance-related* faults are those that cause a slow-down in a component of the system. Examples of our faults in this category include delays caused by thread-sleeps, database table locks, and reduction of the thread and database connection pool sizes. Nineteen of our thirty nine faults fall in this category. *Execution flow-related* faults are those which cause the flow of execution in the system to diverge from the normal case. Our faults in this category include unhandled exceptions in components, runtime component removal,

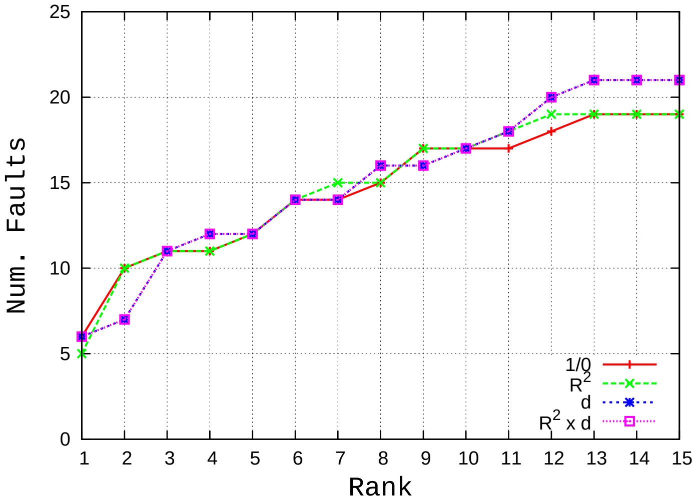


Figure 9.7: Diagnosis with Ratio-score and CR-CS

and corruption of the database authentication credentials. The remaining twenty faults fall in this category.

Figure 9.11 presents the separate evaluation of the two categories of faults. These results indicate that faults affecting the execution flow can be localized more precisely, while faults that affect performance are more difficult to localize. An important reason for the difference in accuracy is that performance-related faults tend to maintain, even strengthen, metric correlations. Consider the example illustrated in Figure 9.12, where the function `foo()` depends on `foo_sub()`, which in turn depends on `foo_sub_sub()`. Because the latter dominates the response time of the two other functions, all the response times are correlated. If because of a fault the response time of `foo_sub_sub()` increases (as shown in the figure), then the response time of the dependent functions will increase as well, thereby maintaining the correlations despite there being a fault.

### 9.6.2 Diagnosis with Alternative Modeling Techniques

Of the alternatives to SLR which we have considered, SLR-T and ARX represent the better choices. As discussed in Chapter 8, though these techniques are costlier and require more stringent tuning to reduce false alarms, they provide better metric and component coverage than SLR. To evaluate whether this extra coverage translates into improved diagnosis accuracy, we use these techniques to model metric correlations and the Max-score, CR-MS algorithm using $d$ as the model-level score. The results, shown in Figure 9.13, show that both modeling techniques im-

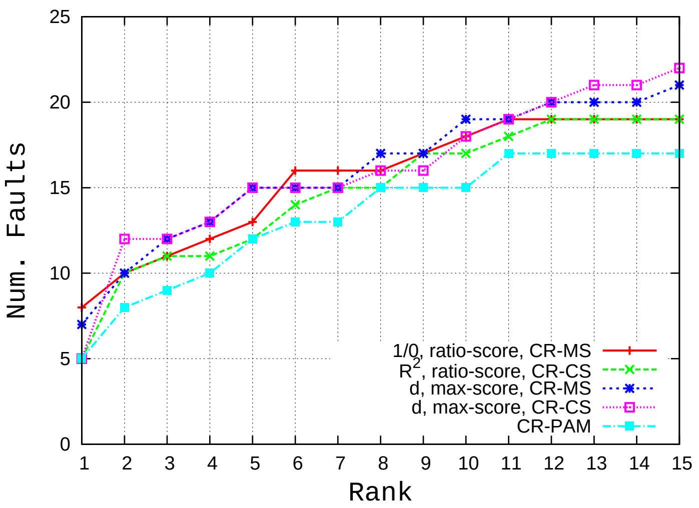


Figure 9.8: Overall comparison of diagnosis methods

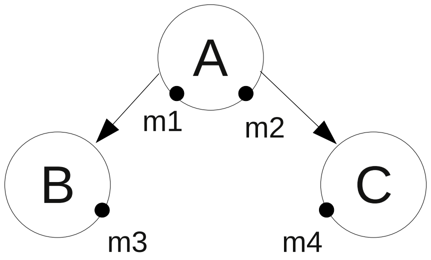


Figure 9.9: Example: component dependencies in a simple system

prove diagnosis accuracy as compared to SLR. While SLR-T brings about a slight improvement, ARX provides considerable gains over both SLR and SLR-T.

To understand the reason behind the improvement, we break down the results with respect to the two fault categories discussed earlier. Figures 9.14 and 9.15 present these results. We observe that for faults that affect execution flow, ARX is slightly better than SLR, which is in turn a little better than SLR-T. For such faults, these more-powerful techniques do not provide any major benefit, as the pertinent correlations are already captured by SLR.

In contrast, Figure 9.15 shows that both alternative modeling techniques perform much better than SLR in diagnosing performance-related faults. The main reason for this improvement lies in the ability of these modeling techniques to represent correlations between activity and response time metrics, which performance-related faults readily perturb. As shown in Chapter 7, both SLR-T and ARX capture a significant number of such correlations.

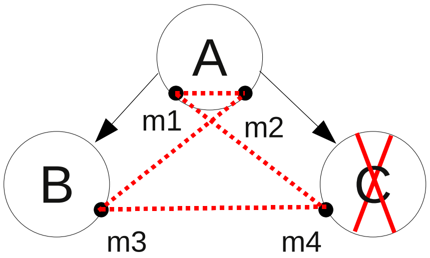


Figure 9.10: Example: component dependencies and broken correlations

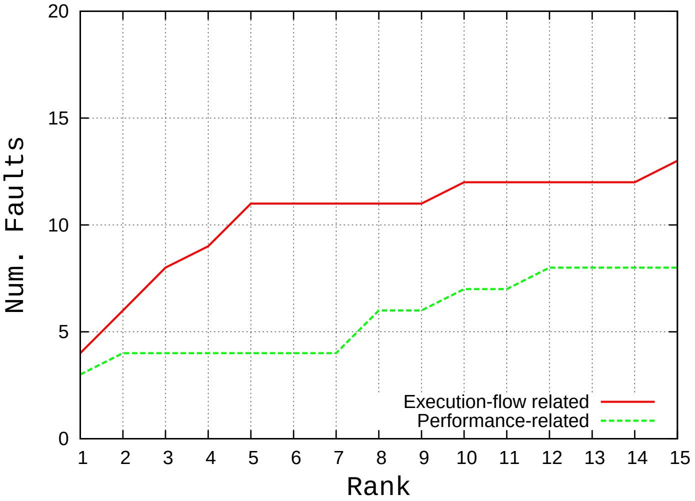


Figure 9.11: Comparison of diagnosis of performance- versus execution flow-related faults

Our results indicate that both SLR-T and ARX improve diagnosis accuracy. Because our system model is an ensemble of correlation models, we can accommodate the unique correlations that these modeling techniques capture. This, however, increases the computational cost of learning our system model. We can reduce this cost by taking the following approach: for each pair of metrics considered, apply the modeling techniques in the order of their cost, and use the technique that has the lowest cost but which is powerful enough to capture the correlation.

As suggested in Chapter 7, we can employ more stringent model identification parameters to reduce the level of false alarms seen with SLR-T and ARX. An alternative approach is to use SLR models for detection at the minimal level as well as for the validation of errors or failures at the detailed level; we can use the combined set of models from all three modeling techniques to perform diagnosis. In Chapter 7, we showed that the three modeling techniques are almost equally good in detecting our faults, which implies that our detection results will not change by

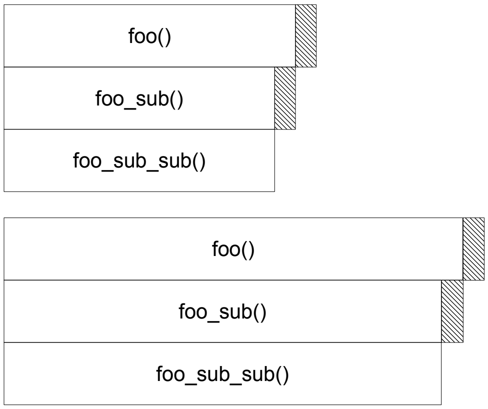


Figure 9.12: Example of a performance fault that does not affect correlations

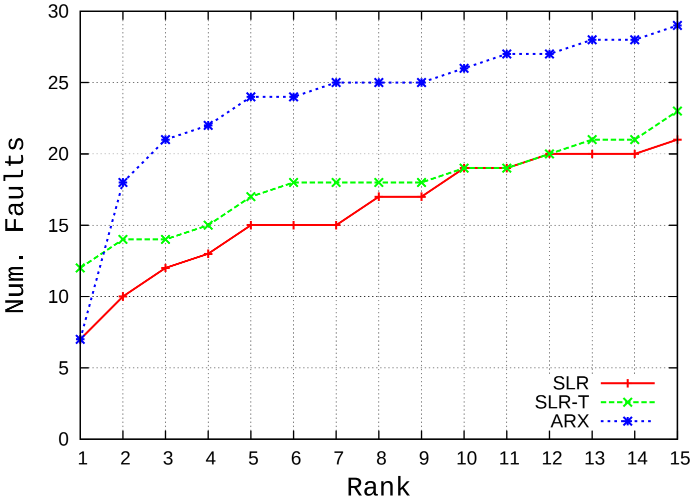


Figure 9.13: Diagnosis with alternative modeling techniques using Max-score, CR-MS

only relying on SLR models. However, our diagnosis will be greatly improved by leveraging all three types of models.

### 9.6.3 Difficulty of Evaluating Diagnosis

Our diagnosis approach produces much more valuable information than what is apparent from our rank-based analysis. Our evaluation is limited in two ways. First, it is stringent in that it relies on strict syntactic matching of component names. Second, in many instances we encounter ties in anomaly scores, which we address by ordering the tied items randomly.

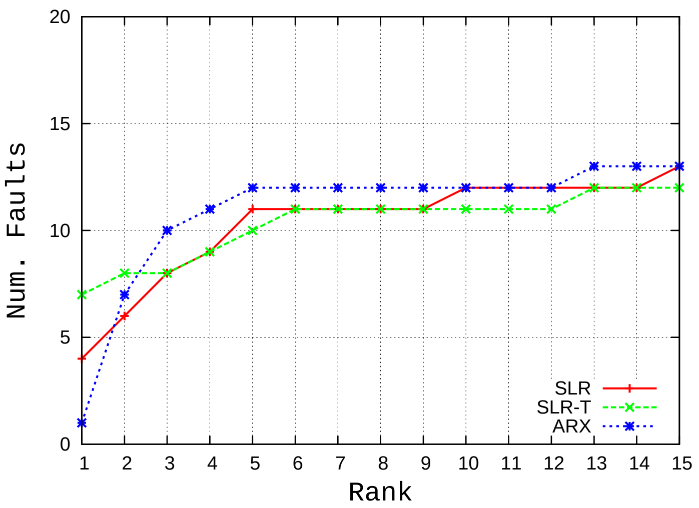


Figure 9.14: Diagnosis with alternative modeling techniques (Execution flow-related faults)

**Syntactic Matching** Strict syntactic matching requires that the exact name of the faulty component be found as a substring in the list of the reported items. For instance, if a fault is injected in the component `OrderEJB`, we require the name of a reported component to contain the `OrderEJB` string. In our experiments, we have commonly observed components that are semantically linked to the faulty component being top-ranked. However, this information is not reflected in our rank-based analysis based on syntactic matching. System operators can often infer fault in a component by semantic matching. For example, if the diagnosis suggests that `TradeEJB.getOrders()` is displaying anomalies, one can readily suspect `OrderEJB` to be possibly faulty. With very little background knowledge, one would know that `OrderEJB` provides access to the order data.

**Ties in Anomaly Scores** When the scores assigned to metrics or components are equal, we order them arbitrarily. As such, a lower rank in some cases does not necessarily indicate diagnosis inaccuracy. Ties are difficult to avoid when working with correlations. With our approach, at the lowest level, we assign anomaly scores to correlation models, which in turn are aggregated at the metric- or component-level. When a correlation model fails, we cannot determine which metric is anomalous; the model's score is shared by the two metrics. This leads to ties. This is especially noticeable when we use Max-score, whereby both the top-ranked and second-ranked items are tied because they are associated with the same model. Therefore, only considering the top 1 metric or component can be misleading; instead, it is better

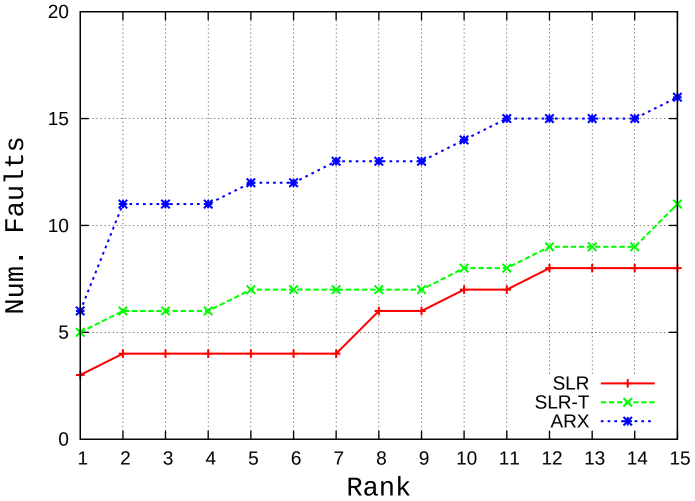


Figure 9.15: Diagnosis with alternative modeling techniques (Performance-related faults)

to take the top $k$ entities into account, where $k > 1$.

An alternative way to reduce the impact of ties is to report correlated metric pairs instead of individual metrics or components to the system operators. While this appears to be a more-natural solution, system operators may find it more difficult to use. As mentioned earlier, the number of correlated pairs reported can be high, making the analysis cumbersome. Moreover, operators may find it difficult to understand and interpret the reported correlations, especially when the correlations are incidental.

## 9.7 Summary

In this chapter we demonstrate that the analysis of metric correlation models can assist system operators in identifying faulty components fast. We devise several anomaly scoring functions to quantify anomalies using the correlation information and the outlier degree observed. We describe techniques to compute metric-level anomaly scores based on the model-level scores. Likewise, we present techniques to generate a component-level diagnosis using the model-level and metric-level scores. We discuss the usefulness of different types of diagnosis information to system operators. We also elaborate on the limitations of metric correlations in pinpointing the exact sources of faults.

We use a multi-tier software system and fault injection experiments to evaluate our diagnosis approach. We perform a detailed analysis of the diagnosis potential of metric correlation models using fine-grained, software component-level faults. Our results indicate that our anomaly scores can assist in finding the faulty component by ranking them high. More specifically, using SLR models, we can shortlist the faulty component in 61% of the cases where errors are detected within the top 15 and 42% within the top 5. Our results suggest that anomaly scores based on the maximum outlier degree observed produce the most accurate diagnosis. We investigate the impact of the nature of faults on the diagnosis accuracy; we show that performance-related faults are harder to identify using SLR models. We study how the more powerful alternatives to simple linear regression can improve diagnosis accuracy: we shortlist the true faulty component in 80% of the cases within the top 15 and 67% within the top 5. Our analysis reveals that the improvement is brought about mostly by capturing correlations between acvitiy and timing metrics.

# Chapter 10

# Discussion

In this chapter we discuss the wider applicability of our solution approach and its limitations. We also present a summary of our work, which is not covered in this thesis, to improve the basic solution approach by relaxing some of its assumptions.

## 10.1 General Applicability

In this work we showed the effectiveness of our solution approach using a test-bed based on the WebSphere application server and the DB2 DBMS. We have shown that stable metric correlations exist in different applications that execute on this setup. Our results should carry over to other Java EE application servers, since the use of the JMX technology to expose management metrics is standard and the Java EE framework itself is standardized. Because the application components implement well-defined interfaces, it is possible to know which metrics are likely important (*e.g.*, those pertaining to remote calls) and thus worth exposing to a monitoring system. The required instrumentation is typically implemented in the middleware, removing any dependence on application-specific instrumentation. The use of a different DBMS should also not affect our results, since different DBMS implement the same core functionality and, as such, and expose similar metrics.

In order to apply our solution approach to other component-based frameworks, such as .Net and CORBA, and to any other type of software systems, three requirements need to satisfied: first, we need a way to discover system components and their metrics; second, we need a way to know what management interfaces are available and how to use them; third, metric collection should be controllable.

The first two requirements can be met by standardization efforts. One prominent effort is Web-Based Enterprise Management (WBEM) [37], which makes use of open standards and technologies to unify the management of distributed computing systems. WBEM leverages the Common Information Model (CIM) [38] to to describe managed entities (hardware or software) in a language-independent formalism and makes this information available through a standard interface. The software vendors need to implement "providers" to allow WBEM infrastructure to interface with their products.

The WBEM infrastructure has been implemented in several popular operating systems including Microsoft Windows, RedHat Enterprise Linux, Mac OS X, and Solaris, allowing WBEM-aware entities running on them to be monitored and managed in a unified manner. WBEM is not specific to enterprise computing systems, but it is being used in other domains such as telecommunications [53]. In the absence of such standardization efforts, we can still apply our approach by discovering components and their metrics from system artifacts such as configuration files, source code, and documentation. We can extract the required information from these artifacts either automatically (*e.g.*, by writing text or code analysis tools) or manually, albeit with more effort.

Knowing the types and the cost of the available metrics can make our solution approach more effective. The use of CIM entails defining metadata that provides some semantic information about metrics. With such information, for example, we can extend the use of threshold-based models in other systems. If we can determine that a metric represents the response time of an entity, we can leverage historical data to automatically create threshold-based models. In the absence of standardized metric metadata, it should also be possible to infer metric types and cost information from naming conventions. It is common for metric names to include terms such as "count", "size", "time", which gives some indication about the nature of the metrics. One avenue for future research is to explore whether the type information can be inferred automatically by analyzing the nature of the metric data (*e.g.*, by considering the values assumed, their variance, *etc.*.)

The third requirement is easily met in Java EE-based systems because much of the useful instrumentation is implemented in the middleware. In other systems, however, the developers need to add the instrumentation that is likely to be relevant to understanding the behaviour and performance of the system. In addition to instrumenting the system and providing means to access the exposed data, the developers also need to provide mechanisms for turning the instrumentation on and off.

It should be noted that most software systems have instrumentation to aid monitoring In addition, knobs to control the activation of this instrumentation may exist to help reduce the performance overhead. In these cases, what is needed is to integrate what already exists into a WBEM-like infrastructure to make it seamless to implement our solution approach. Often libraries exist to make this integration easy; for instance, this is the case for Windows Management Instrumentation (WMI), which is an implementation of WBEM on the Microsoft Windows Platform.

A .Net-based system would be an appropriate target to validate our claim that our solution approach applies to other systems because the .Net framework is widely used in practice and is supported by a WBEM infrastructure. This work would require choosing a target application and supporting components (*e.g.*, a DBMS), ensuring that the system is instrumented to expose application-level metric data through WMI, using WMI to retrieve the metric and component metadata, and periodically collect the metric data to identify and model correlations.

## 10.2 Limitations

The monitoring and problem-determination approaches described in this thesis have some limitations. These arise from the constraints imposed by the solution requirements and our choice of the solution approach. In this section we discuss these limitations, and where possible suggest ways of addressing them.

**Learning faulty behaviour:** One important limitation of our system modeling approach is that faults can become part of the learned system behaviour. The metric data needed to build the system model should ideally come from the target system while it executes fault-free. If this is not the case, we will learn a system's faulty behaviour as though it were normal. Fault-free systems do not exist in general. However, faults can be benign, whereby they do not affect a system's reliability. System operators need to make sure that during the period of metric data collection, the system was not subject to any apparent fault.

**Multiple, independent faults:** Our monitoring approach allows the detection of multiple independent faults, provided these faults affect the metrics we model. However, the accuracy of our diagnosis approach may suffer in the presence of multiple faults, since the effects of these faults on the metric correlations are likely to be confounded. Our current approach does not address this problem explicitly.

Nevertheless, if multiple faults exist, the iterative detection and resolution of the faults will improve the diagnosis accuracy of the remaining faults.

**Faults with subtle effects and transient faults:** Our monitoring approach relies on statistical tests to detect anomalies; these tests detect significant deviations from the modeled norm. As a result, it is difficult to detect faults that cause subtle changes in the system. Nonetheless, such faults tend to become more severe as time passes or as their effects spread, and thus they are likely to be detected eventually. For example, a memory leak may not be noticeable for a long period, but will eventually cause excessive page faults, even thrashing. In addition, our approach does not work well for detecting faults that are transient and short-lived. While we may be able to detect an instance of the fault manifestation, by the time detailed monitoring is enabled, the fault may become inactive. Such faults are better detected using alternative approaches such as expectation-based tracking [120].

**Fault-induced correlations:** Our error detection approach is predicated on the idea that correlations break because of faults. However, faults can also induce new correlations that were not apparent before. Consider, for example, the case of a faulty link between a web server and an application server. This may delay communication between the two servers. Before the fault, the latency between the two servers may not have been a relevant factor. But, as a result of the fault, the latency becomes the most important contributor to the overall response time.

In order to capture newly induced correlations, we need to analyze all the available metrics, not only those associated with correlations. One relevant approach mentioned in Chapter 3 is proposed in [21, 22]. This approach employs multivariate statistical modeling and data reduction techniques to track all the available metrics. However, further research is needed to see if such a modeling approach can be used with adaptive monitoring.

**Metrics not covered by correlations:** Our approach involves identifying correlations between metrics and representing them using regression models. Our correlation models cannot cover all system metrics. First, a metric can be related to more than one other via complex relationships. Capturing such correlations would require a multi-variate model, which is not addressed in our work. Discovering multi-variate models is expensive; the cost of a naïve search is exponential in the number of the metrics per considered model. Second, even if a metric is correlated to another metric, the modeling techniques chosen may not be flexible enough to capture the metric relationship.

For metrics not captured by correlation models, we could use threshold-based models or similar, single-variable alternative models. Automatically creating these models and configuring them to achieve the right balance between sensitivity to faults and susceptibility to false alarms is non-trivial.

**Critical metrics:** Our monitoring approach is agnostic to the metric semantics. In particular, it considers all metric correlations to be equally important, regardless of how critical they are to system operators. However, certain metrics are more important than others, especially in the business context. For example, compliance with performance SLOs can be critical for some services. In this case, a top-down approach may be more appropriate, whereby the key metrics are chosen first, followed by those that affect them. Breitgand *et al.* [16] investigates an approach whereby thresholds for component metrics are derived automatically from SLOs.

## 10.3 Extending the Basic Solution Approach

In collaboration with colleagues, we have investigated several ideas to improve the correlation-based and adaptive monitoring approaches. These efforts have resulted in several publications, which we summarize briefly below:

**Monitoring with metric correlations in clustered environments:** In [104], we studied the use of metric correlations to monitor clustered systems, in which some subsystems (*e.g.*, application servers) are replicated. In that study, we observed that a blind approach to learning metric correlations is not necessarily effective; instead, making use of the high level structure (*i.e.*, the topology) of a system not only helps reduce the cost of identifying metric correlations, but also reduces the likelihood of retaining less stable correlation models. The study suggests that comparing the analysis of metric correlations on different peers can help isolate the cause of observed anomalies.

**Tracing-augmented adaptive monitoring:** In [103], we extended our basic adaptive monitoring approach, including trace-based analysis to achieve higher diagnosis accuracy. We propose a three-step adaptive monitoring approach. At the minimal level, key system metrics are monitored by means of thresholds. When anomalies are detected, an extended set of metrics is collected and checked using correlation models. If the anomalies are corroborated, we enable the collection of ARM request traces, which incur higher performance overhead. The analysis of the metric correlations allows us to identify the likely faulty replica in a clustered system. Traces collected from the faulty replica are compared with those of its healthy peers; components involved in the execution of user requests are then ranked according to the degree to which their behaviour and performance deviate from the normal case.

**Using information theory to model metric correlations:** In [69], we investigated the use of mutual information, an information-theoretic measure, to capture metric correlations. Mutual information obviates the need to specify *a priori* a fixed functional form (*e.g.*, simple linear regression) By capturing metric correlations irrespective of the form of the underlying relationships, we can improve the coverage of system dynamics. We propose an efficient method to track the correlated metrics. We cluster the correlated metrics into groups, and we track the groups using the entropy of the normalized metric values in each group. Tracking at the group level is much more efficient than tracking a large set of individual correlation models.

**Addressing heteroscedasticity in metric correlations:** In [71], we show that for many pairs of correlated metrics in complex software systems, the variance of the predicted variable is not constant. This behaviour violates the assumptions of linear regression, making the correlations modeled with linear regression less effective for monitoring. In particular, for many metric pairs, we have observed that the variance of the residuals increases with the predictor variable. To address this problem, we employ the method of generalized least squares with linear regression to account for the non-constant residual variance. We show through experiments that this variant can capture many metric correlations more effectively by considering the changing residual variance.

**Identifying three-variable metric correlations** In [72], we exploit the heteroscedasticity phenomena observed in many two-variable relationships to discover three-variable models. One common reason that underlies heteroscedasticity is a variable missing from the model. We thus perform a search for three-metric models for only those metric pairs which suffer from heteroscedasticity. Our approach keeps the cost within $O(n^2)$ for $n$ metrics, whereas a naïve search for three-variable models would cost $O(n^3)$.

**Improving correlation-based diagnosis using structure information** In [70], we extend the work presented in [69] and present a diagnosis algorithm to locate faulty components which incorporates knowledge of component dependencies. We show that diagnosis accuracy can be improved significantly by leveraging information about the system's structure; this information need not be complete or perfect to be useful.

**Learning fault signatures based on metric correlations** As discussed in Chapter 9, a number of factors can reduce the diagnosis accuracy of methods based on metric correlations. If faults are known beforehand, an alternative fault identification approach is to find the unique set of perturbed correlations associated with the faults. In [68], we study the problem of identifying the most pertinent metric correlations in order to detect known, recurrent faults and to diagnose them accurately, while requiring a minimal number of correlation models. We propose a methodology to find the relevant metric correlations using neural networks.

# Chapter 11

# Conclusions and Future Research

In this thesis, we tackled the challenge of overseeing complex software systems in an automated and cost-effective manner. The motivation behind this effort was to reduce the cost of monitoring and problem determination both in terms of human resources and the monitoring overhead. To this effect, we devised an automated, adaptive monitoring approach based on management metrics. Our approach entails modeling and monitoring complex software systems using simple, efficient statistical techniques using metric data alone, without domain knowledge, detailed information about system structure and its inner workings, and *a priori* knowledge of faults. Further, our approach involves changing the set of management metrics that are tracked, in order to fulfill the information needs of the task at hand. The automated nature of our solution approach allows the cost human resources to be reduced, and the adaptive capability allows the monitoring overhead to be minimized. Our approach can be implemented easily and deployed with little or no change to the the target systems. We validated our solution approach using a realistic test-bed, showing its ease of implementation, efficiency, and effectiveness.

Our approach alleviates the burden of modeling and tracking the health of complex software systems on human operators and experts. Although human operators are ultimately responsible for resolving problems that arise in these systems, our approach accelerates the process, allowing them to save valuable time. Therefore, the cost system monitoring can be kept minimal, allowing resources to be allocated to providing the core system functionality.

Our work offers a positive outlook on the future of system management, in particular monitoring. Our research demonstrates that simple statistical and machine-learning techniques can enable larger, more complex software systems to be monitored effectively and with little or no human involvement. A distinguishing contribution of this work is to show that these benefits can be had without sacrificing system performance.

While our work represents a significant step in the pursuit of fully automated, adaptive software system monitoring, it also brings to light a number of challenges, which future work needs to address. We discuss some of these challenges next.

## 11.1 System Modeling

System modeling is the most critical building block to enable automated monitoring. Two aspects of our approach to system modeling that can be improved are described below.

**Identifying metric correlations:** The adaptive monitoring solution described in this work is designed to keep the monitoring overhead low during live monitoring. Prior to the monitoring stage, we need to identify and model the stable metric correlations. This requires that all metrics be collected over a period long enough to record the representative behaviour of the system. As discussed in Chapter 6, the cost of monitoring all the available metrics can be high and thus cannot be enabled for long periods in a production system. One solution is to use a complete replica of the production system to identify the correlations using real-world workload traces. This solution, however, is in general impractical because of its cost.

Two alternative solutions that we have considered are described below. Their experimental validation is part of our future work.

- In clustered systems, where replicas of subsystems exist, full monitoring can be enabled in part of the system. For example, if there are multiple web and application servers to handle the workload, metrics can be collected from one web server, one application server, and the back-end. The system can be configured to direct representative but less work to the monitored part to compensate for the reduced efficiency. If the replicas are similar and the workload is balanced fairly among them, we can expect the same correlations to be discovered on the non-monitored counterparts. If replicas are different (*e.g.*, based on different hardware or system software), then an incremental approach to identifying the correlations is needed. Different parts of the system can be monitored at different times and the correlation identification process will end when all parts have been covered. The main shortcomings of this approach include the need to make load-balancing in the system monitoring-aware and, if needed, to implement the incremental model-learning logic.

- The idea of incremental learning can be further extended to work at the level of metric subsets. We can specify a performance overhead budget within which correlation identification has to take place. In a series of phases, we can identify stable correlations by exploring subsets of the metrics at a time. In each phase, a subset of available metrics is chosen to be enabled for collection. These metrics are collected for a specified period, after which correlations between metrics are assessed for strength and stability. In the subsequent iteration, another subset of metrics is enabled and analyzed, while the collection of those metrics that are no longer needed is disabled. These phases continue as long as the space of metric pairs to explore is not exhausted.

  The downside to the incremental system modeling approach is the much longer time needed to find all stable correlations as compared to the approach where all data is collected and modeling applied at once.

**Multi-scale metric correlations:** In this thesis we assumed that metrics are read periodically at fixed intervals. The time interval is set so that the dynamics of interest are reflected readily in the collected data. It is, however, possible for metric correlations not to be strong and stable at the chosen resolution, but still be relevant at a larger time scale. We would like to investigate correlations at multiple time scales and study whether correlations at coarser time scales can improve system monitoring.

## 11.2 Fine-Grained Adaptive Monitoring

One challenge for future work with respect to adaptive monitoring is to make it more fine-grained. In this thesis we devised an approach where two monitoring levels (and, possibly, a third intermediate level) are pre-specified, and adaptation takes place by moving from one level to another when needed.

An adaptive approach that works at the level of individual metrics will be more efficient. Two main challenges in this respect are as follows: first, we need to determine when enough metrics have been analyzed to determine the existence of faults reliably and their likely source; second, we need to find ways to obtain this information with the least cost. Fine-grained adaptation may not be possible without some knowledge of the system structure . Further, knowledge of the cost of collecting individual metrics requires information about metric types. As discussed in Chapter 10, this information may be available through standardized metric metadata or be inferred from system artifacts.

## 11.3 Diagnosis

Opportunities for improvement also exist for our diagnosis approach. Some ideas to this effect are presented next.

**Use of information about system structure:** In this thesis we have not used any information about how the target system is structured. However, such information can help reduce the cost of modeling the system and improve diagnosis. As mentioned in Chapter 3, this information can often be inferred from system artifacts or the monitoring data. With knowledge of the high-level topology of the system (*i.e.*, information about how the subsystems are connected), we can drastically reduce the number of metric combinations whose correlations need to considered. Instead of performing cross-correlation of all the metrics exposed by a system, we only need to correlate metrics of subsystems that are connected.

Knowledge of the system structure can help refine the diagnosis based on metric correlations. One way to leverage this information is to distinguish the correlations that arise because of direct component dependencies from those that are incidental. We can analyze the correlation graph created using the causal correlations to identify points at which correlations break when a fault occurs. Comparing the results of the diagnosis approach presented in Chapter 9 and this graph-based approach, and studying whether the two approaches are useful together, needs to be explored.

One difficulty with trying to identify faulty components based on broken correlations is that components that depend on the true faulty components often behave as though they were faulty (*i.e.*, they are associated with broken correlations). Knowing the structure, we could undo this effect by adjusting the anomaly scores such that components that are not dependent on a reported anomalous component are assigned larger scores than those that are. While an idea to this effect has been explored by Agarwal *et al.*. [1], further work is needed to see how easily and how well the inaccuracies arising because of dependencies can be undone.

**Confidence score for the diagnosis:** The diagnosis produced by our monitoring system consists of a list of components or their metrics prioritized according to assigned anomaly scores. The system operators can process this list in order by checking the status of the reported components. Our results in Chapter 9 show that very often the faulty component is included in the top-ranked components. Nevertheless, sometimes the true faulty component is not included in the reported list or it ranks too low. In such a case, the system operators will likely waste much time going through the reported list, even though it may not be useful. Therefore, one challenge that needs to be addressed is to devise a measure of confidence in the reported diagnosis. Such a confidence score should be high when the true faulty component is included in the reported list and it ranks high, and it should be low otherwise.

A number of factors could be considered to compute such a confidence score, including the quality of the metric correlation models used to produce the diagnosis, the number of independent models involved (*i.e.*, models that do not share metrics), and the number of different clusters of correlated metrics to which the anomalous metrics pertain (*i.e.*, an estimate of the number of affected dynamics in the system).

**Noise in the presence of faults:** Metric correlations are not equally robust to changes in the system. Correlations that arise because of dependencies in the system are robust to changes in the load or resource availability. These correlations are affected when the dependencies or the underlying structure of the system changes. On the other hand, other correlations, in particular the models that capture them, are sensitive to the configuration of the system. For example, a correlation between a metric that tracks activity and a metric that measures response time in a component may be sensitive to the amount of CPU and memory resources allocated. This resource configuration may not change for long periods of time, and thus it is reasonable to identify and use this correlation for monitoring purposes. However, problems arise when faults occur in the system; if the faults affect resource availability, the parameters of the correlation may change. Even though the correlation may still exist, it displays different characteristics than when it was identified. The consequence is that some correlation models may report outliers despite the fact that the correlations still hold, albeit displaying different characteristics.

In this work we address this problem by using both the Studentized residuals and the relative absolute residual (see Section 7.2.3 in Chapter 7. The latter is less sensitive to small changes in the values of metrics that are strongly correlated. Although, this technique addresses the problem to some extend, a deeper investigation of this issue is needed.

### 11.3.1 Correlation-Friendly Instrumentation

From our work, we learned a number of lessons on how to instrument software systems to make correlation-based monitoring and diagnosis more effective. First, code should be instrumented such that any regularity or symmetry in the behaviour or performance of the logic therein is exposed. The following example illustrates this point. It is common for developers to instrument functions such that information about successful completion is captured. Listing 2.1 in Chapter 2 is an example of such a function, whereby a counter tracks the number of completions just before the function exits. If a fault prevents this function from completing, the counter alone may not indicate an anomaly. Moreover, if the call to this function is nested in several layers of function calls, where the calling functions also have completion counters before their exit point, the correlations between the counters will not break, since their values will drop at the same time. Instead, by having counters both at the entry and the exit points, an anomaly can readily be detected inside each function, including the function that is directly affected by the fault.

Studying the kinds of regularity that exist in software systems, how instrumentation can capture them in an efficient way, and how such instrumentation can improve our solution approach remains to be studied.

Second, exposing metrics that reflect the state of the system (*e.g.*, number of active entities, number of concurrent operations, *etc.*.) can be effective in detecting and helping localize performance faults. Not all instrumentation costs the same; obtaining timing information, for example, is expensive. As such, exposing state data, especially data that reflects regularity pertaining to the state, provides an efficient alternative to detecting and diagnosing performance faults with timing metrics.

### 11.3.2 Other Applications of Metric Correlations

In this thesis we explored the error detection and diagnosis potential of metric correlations. Metric correlations can serve other purposes such as assessing the impact of failures or errors in the system. For example, by analyzing the extent to which metric correlations are perturbed, one may estimate the degree to which the system is affected by a fault. Similarly, by analyzing the changes in correlations, one can study how a fault affects the system. Such impact analysis need not be limited to failure and errors; it can be used to study the impact of changes to the system, including its software. For instance, strengthening of existing correlations and appearance of new ones may indicate that the change made to the system introduced a new bottleneck.

## Bibliographical Notes

The work presented in this thesis builds on ideas published by the author in [105, 108, 109, 110, 111]. The correlation-based monitoring approach is investigated further by the author in [103, 104, 107]. In [103, 107], correlation-based monitoring is combined with the analysis of log files and traces. In [104], correlation-based monitoring is applied to clustered, multi-tier software systems. Extensions and related work to which the author has contributed comprise [68, 69, 70, 71, 72]. Finally, some novel ideas which are being explored are described in [106].

# References

[1] M. K. Agarwal, K. Appleby, M. Gupta, G. Kar, A. Neogi, and A. Sailer. Problem determination using dependency graphs and run-time behavior models. In *Proceedings of the 15th IFIP/IEEE Distributed Systems: Operations and Management (IM)*, Davis, California, USA, November 2004. 32, 158

[2] Manoj Agarwal, Nikos Anerousis, Manish Gupta, Vijay Mann, Lily Mummert, and Narendran Sachindran. Problem determination in enterprise middleware systems using change point correlation of time series data. In *Proceedings of the IFIP/IEEE Network Operations and Management Symposium (NOMS)*, April 2006. 30

[3] Manoj K. Agarwal, Narendran Sachindran, Manish Gupta, and Vijay Mann. Fast extraction of adaptive change point based patterns for problem resolution in enterprise systems. In *DSOM '06: Proceedings of the 17th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management*, pages 161–172, 2006. 31

[4] Sandip Agarwala, Yuan Chen, Dejan Milojicic, and Karsten Schwan. QMON: QoS- and utility-aware monitoring in enterprise systems. In *Proceedings of the 3rd IEEE International Conference on Autonomic Computing (ICAC)*, 2006. 35

[5] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Proceedings of the 19th ACM symposium on Operating systems principles (SOSP)*, pages 74–89, New York, NY, USA, 2003. ACM Press. 27

[6] Jennifer M. Anderson, Lance M. Berc, Jeffrey Dean, Monika R. Henzinger, Shun tak A. Leung, L. Sites, Mark T. Vandevoorde, Carl A. Waldspurger,

William E. Weihl, L. M. Berc, S. Ghemawat, M. R. Henzinger, and S. t. A. Leung. Continuous profiling: Where have all the cycles gone? In *ACM Transactions on Computer Systems*, pages 1–14, 1997. 34

[7] Apache Software Foundation. Apache Module mod_status. http://httpd.-apache.org/docs/2.0/mod/mod_status.html. 19

[8] Matthew Arnold and Barbara G. Ryder. A framework for reducing the cost of instrumented code. *ACM SIGPLAN Notices*, 36(5):168–179, 2001. 34

[9] Chris Atkeson, Andrew Moore, and Stefan Schaal. Locally weighted learning. *AI Review*, 11:11–73, April 1997. 81, 85

[10] 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. 10

[11] Victor Bahl, Paul Barham, Richard Black, Ranveer Chandra, Moises Goldszmidt, Rebecca Isaacs, Srikanth Kandula, Lun Li, John MacCormick, David A. Maltz, Richard Mortier, Mike Wawrzoniak, and Ming Zhang. Discovering dependencies for network management. In *Proceedings of the fifth Workshop on Hot Topics in Networks (HotNets-V)*, 2006. 27

[12] Paul T. Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using magpie for request extraction and workload modelling. In *Proc. of the 6th Symposium on Operating System Design and Implementation (OSDI)*, pages 259–272, 2004. 27

[13] Paul T. Barham, Rebecca Isaacs, Richard Mortier, and Dushyanth Narayanan. Magpie: Online modelling and performance-aware systems. In *Proceedings of HotOS'03: 9th Workshop on Hot Topics in Operating Systems (HotOS)*, pages 85–90, 2003. 27

[14] Peter Bodik, Greg Friedman, Lukas Biewald, Helen Levine, George Candea, Kayur Patel, Gilman Tolle, Jon Hui, Armando Fox, Michael I. Jordan, and David Patterson. Combining visualization and statistical analysis to improve operator confidence and efficiency for failure detection and localization. In *Proc. of the 2nd IEEE International Conference on Autonomic Computing (ICAC)*, Seattle, June 2005. 28, 32

[15] G. Bonanno, G. Caldarelli, F. Lillo, S. Micciche, N. Vandewalle, and R. N. Mantegna. Networks of equities in financial markets. *The European Physical Journal B*, 38:363–371, 2004. 103

[16] David Breitgand, Ealan Henis, and Onn Shehory. Automated and adaptive threshold setting: Enabling technology for autonomy and self-management. In *ICAC '05: Proceedings of the Second International Conference on Automatic Computing*, pages 204–215, Washington, DC, USA, 2005. IEEE Computer Society. 152

[17] A. Brown, G. Kar, and A. Keller. An active approach to characterizing dynamic dependencies for problem determination in a distributed environment. In *Proceedings of IFIP/IEEE International Symposium on Integrated Network Management*, pages 377–390, May 2001. 27, 30

[18] 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. 3

[19] 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. 17, 33, 35

[20] 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. 12, 13, 22

[21] Haifeng Chen, Guofei Jiang, Cristian Ungureanu, and Kenji Yoshihira. Failure detection and localization in component based systems by online tracking. In *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 750–755, 2005. 28, 29, 32, 151

[22] Haifeng Chen, Guofei Jiang, Cristian Ungureanu, and Kenji Yoshihira. Combining supervised and unsupervised monitoring for fault detection in distributed computing systems. In *Proceedings of the 2006 ACM symposium on Applied computing (SAC)*, pages 705–709, New York, NY, USA, 2006. ACM Press. 29, 32, 151

[23] M. Chen, A. Zheng, J. Lloyd, M. Jordan, and E. Brewer. Failure diagnosis using decision trees. In *Proceedings of the International Conference on Autonomic Computing*, New York, NY, 2004. 27, 32

[24] Mike Y. Chen, Anthony Accardi, Emre Kiciman, Jim Lloyd, Dave Patterson, Armando Fox, and Eric Brewer. Path-based failure and evolution management. In *Proceedings of the International Symposium on Networked Systems Design and Implementation (NSDI*, pages 309–322, 2004. 2, 27

[25] Mike Y. Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric A. Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *International Conference on Dependable Systems and Networks (DSN)*, pages 595–604, 2002. 27, 32

[26] 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 *Proceedings of the sixth Symposium on Operating Systems Design and Implementation (OSDI)*, pages 231–244, December 2004. 31, 32

[27] Ira Cohen, Steve Zhang, Moisés Goldszmidt, Julie Symons, Terence Kelly, and Armando Fox. Capturing, indexing, clustering, and retrieving system history. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, pages 105–118, 2005. 31

[28] Jacob Cohen, Patricia Cohen, Stephen G. West, and Leona S. Aiken. *Applied Multiple Regression/Correlation Analysis for the Behavioral Sciences.* Lawrence Erlbaum, 2nd edition, 2003. 75, 76, 77, 83

[29] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to algorithms.* McGraw-Hill Book Company, Cambridge, London, 2. edition, 2001. 103

[30] William H. Crown. *Statistical Models for the Social and Behavioral Sciences: Multiple Regression and Limited-Dependent Variable Models.* Greenwood Publishing Group, 1998. 75

[31] Peter J. Denning and Jeffrey P. Buzen. The operational analysis of queueing network models. *ACM Computing Surveys*, 10(3):225–261, 1978. 62

[32] Ada Diaconescu, Adrian Mos, and John Murphy. Automatic performance management in component based software systems. In *ICAC 04: Proceedings of the First International Conference on Autonomic Computing (ICAC04*, pages 214–221. IEEE Computer Society, 2004. 34

[33] Y. Diao, J. L. Hellerstein, S. Parekh, and J. P. Bigus. Managing web server performance with autotune agents. *IBM Systems Journal*, 42(1):136–149, 2003. 25

[34] Yixin Diao. Stochastic modeling of Lotus Notes with a queueing model. In *International Computer Measurement Group Conference*, pages 229–238, 2001. 25

[35] 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 *IFIP/IEEE 8th International Symposium on Integrated Network Management (IM)*, pages 157–170, 2003. 28

[36] John Dilley, Rich Friedrich, Tai Jin, and Jerome A. Rolia. Measurement tools and modeling techniques for evaluating web server performance. In *Proceedings of the 9th International Conference on Computer Performance Evaluation: Modelling Techniques and Tools*, pages 155–168, London, UK, 1997. Springer-Verlag. 25

[37] Distributed Management Task Force, Inc. Web-Based Enterprise Management (WBEM). http://www.dmtf.org/standards/wbem/. 22, 149

[38] Distributed Management Task Force, Inc. Common Information Model Standards. http://www.dmtf.org/standards/cim/. 149

[39] Mikhail Dmitriev. Profiling Java applications using code hotswapping and dynamic call graph revelation. In *Proceedings of the 4th international workshop on Software and performance (WOSP)*, pages 139–150, New York, NY, USA, 2004. ACM Press. 33, 35

[40] Armando Fox and David Patterson. Self-repairing computers. *Scientific American*, June 2003. 3

[41] Eibe Frank, Leonard Trigg, Geoffrey Holmes, and Ian H. Witten. Technical note: Naïve Bayes for regression. *Machine Learning*, 41(1):5–25, 2000. 31

[42] Zhenghua Fu and Nalini Venkatasubramanian. Adaptive parameter collection in dynamic distributed environments. In *IEEE International Conference on Distributed Computer Systems (ICDCS)*, 2001. 34, 36

[43] Saeed Ghanbari and Cristiana Amza. Semantic-driven model composition for accurate anomaly diagnosis. In *International Conference on Autonomic Computing*, 2008. 31

[44] Dan Gunter and Brian Tierney. NetLogger: A toolkit for distributed system performance tuning and debugging. In *Proc. of the IFIP/IEEE Eighth International Symposium on Integrated Network Management (IM 2003)*, pages 97–100, March 2003. 22, 34

[45] Zhen Guo, Guofei Jiang, Haifeng Chen, and Kenji Yoshihira. Tracking probabilistic correlation of monitoring data for fault detection in complex systems. In *International Conference on Dependable Systems and Networks (DSN)*, pages 259–268, 2006. 29

[46] Manish Gupta, Anindya Neogi, Manoj K. Agarwal, and Gautam Kar. Discovering dynamic dependencies in enterprise environments for problem determination. *Lecture Notes in Computer Science*, 2867:221–233, Jan 2003. 27

[47] Jiawei Han and Micheline Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 2nd edition, 2006. 101

[48] 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. 30

[49] Daniel E. Hecker. Occupational employment projections to 2014. *Monthly Labor Review*, pages 70–101, November 2005. 2

[50] Joseph L. Hellerstein, Fan Zhang, and Perwez Shahabuddin. Characterizing normal operation of a web server: Application to workload forecasting and problem detection. In *Proceedings of Computer Measurement Group*, December 1998. 28

[51] Edwin A. Hernandez, Matthew C. Chidester, and Alan D. George. Adaptive sampling for network management. *Journal of Network and Systems Management*, 9(4):409–434, 2001. 34, 36

[52] Hewlett-Packard Development Co. HP OpenView Management Software. http://www.managementsoftware.hp.com/. 23, 106

[53] Chris Hobbs. *A Practical Approach to WBEM/CIM Management*. Auerbach Publications, 2004. 149

[54] IBM Corporation. Autonomic Computing. http://www.research.ibm.com/-autonomic/. 3

[55] 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. 19

[56] 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. 19

[57] IBM Corporation. IBM Tivoli Software. http://www.ibm.com/software/-tivoli/. 23, 34, 106

[58] IBM Corporation. WebSphere Application Server. http://www.ibm.com/-software/webservers/appserv/. 49

[59] IBM Corporation. DB2 Universal Database. http://www.ibm.com/software/-data/db2/udb/. 49

[60] IBM Corporation. PlantsByWebSphere Sample. http://www.ibm.com/-developerworks/websphere/library/samples/plantsby.html. 50

[61] IBM Corporation. Trade. http://www-01.ibm.com/software/webservers/-appserv/benchmark3.html. 51

[62] IBM Corporation. WebSphere Application Server, Monitoring System Health. http://publib.boulder.ibm.com/infocenter/wasinfo/v6r0/topic/-com.ibm.websphere.express.doc/info/exp/ae/tprf_monitoringhealth.html. 98

[63] IBM Corporation. IBM WebSphere Application Server V6 Performance Tools. http://publib.boulder.ibm.com/infocenter/ieduasst/v1r1m0/topic/-com.ibm.iea.was_v6/was/6.0/Performance/WASv6_PerformanceTools.pdf. 33

[64] JBoss Enterprise. A Framework for Organizing Cross Cutting Concerns. http://jboss.org/jbossaop/. 18

[65] G. Jiang, H. Chen, and K. Yoshihira. Discovering likely invariants of distributed transaction systems for autonomic system management. In *Proceeding of the International Conference on Autonomic Computing*, 2006. 29

[66] Guofei Jiang, Haifeng Chen, and Kenji Yoshihira. Modeling and tracking of transaction flow dynamics for fault detection in complex systems. *IEEE Trans. Dependable Sec. Comput.*, 3(4):312–326, 2006. 29, 33

[67] Guofei Jiang, Haifeng Chen, and Kenji Yoshihira. Efficient and scalable algorithms for inferring likely invariants in distributed systems. *IEEE Transactions on Knowledge and Data Engineering*, 19(11):1508–1523, 2007. 30, 71

[68] 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. 154, 162

[69] 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. 153, 154, 162

[70] 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. 154, 162

[71] 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. 153, 162

[72] 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. 153, 162

[73] Mark W. Johnson. Monitoring and diagnosing applications with ARM 4.0. In *Proceedings of the Computer Measurement Group (CMG) Conference*, pages 473–484, 2004. 19, 27, 34

[74] Jupiter Research. Retail Web Site Performance: Consumer Reaction to a Poor Online Shopping Experience. http://www.akamai.com/4seconds. 106

[75] Leonard Kaufman and Peter J. Rousseeuw. *Finding groups in data: an introduction to cluster analysis.* Wiley Series in Probability and Mathematical Statistics. Applied Probability and Statistics, New York, 1990. 115

[76] J.O. Kephart and D.M. Chess. The vision of Autonomic Computing. *IEEE Computer*, 36(1):41–50, January 2003. 2, 3

[77] Emre Kiciman. *Using statistical monitoring to detect failues in Internet services*. PhD thesis, Stanford University, 2005. 24

[78] 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. 27, 32, 33

[79] Emre Kiciman and Ben Livshits. AjaxScope: A platform for remotely monitoring the client-side behavior of web 2.0 applications. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, 2007. 35

[80] Emre Kiciman and Helen Wang. Live Monitoring: Using adaptive instrumentation and analysis to debug and maintain web applications. In *Proceedings of the 11th Workshop on Hot Topics in Operating Systems (HotOS)*, San Diego, CA, May 2007. 35

[81] Abdelkader Lahmadi, Laurent Andrey, and Olivier Festor. On the impact of management on the performance of a managed system: A JMX-based management case study. In *Proceedings of the 16th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM), Ambient Networks*, pages 24–35, 2005. 33

[82] Lawrence Berkeley National Laboratory. The Internet Traffic Archive. http://ita.ee.lbl.gov/html/traces.html. 49

[83] Jun Li. Monitoring and characterization of component-based systems with global causality capture. In *Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS)*, pages 422–31, May 2003. 27

[84] Lei Li, Kalyanaraman Vaidyanathan, and Kishor S. Trivedi. An approach for estimation of software aging in a web server. In *Proceedings of the International Symposium on Empirical Software Engineering (ISESE)*, pages 91–100, 2002. 26

[85] Lennart Ljung. *System Identification - Theory For the User.* PTR Prentice Hall, Upper Saddle River, N.J., 2nd edition, 1999. 80

[86] Ying Lu, Tarek Abdelzaher, Chenyang Lu, Lui Sha, and Xue Liu. Feedback control with queueing-theoretic prediction for relative delay guarantees in web servers. In *Proceedings of the 9th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Washington, DC, USA, 2003. IEEE Computer Society. 25

[87] Michael R. Lyu, editor. *Handbook of software reliability and system reliability.* McGraw-Hill, Inc., Hightstown, NJ, USA, 1996. 11

[88] Miroslaw Malek, Felix Salfner, and Günther Hoffmann. Self rejuvenation: An effective way to high availability. In *International workshop on Self-* Properties in Complex Information Systems*, 2004. 30

[89] Rosario N. Mantegna. Hierarchical structure in financial markets. *The European Physical Journal B*, 11(1):193–197, 1999. 103

[90] Matthew L. Massie, Brent N. Chun, and David E. Culler. The Ganglia distributed monitoring system: Design, implementation, and experience. *Parallel Computing*, 30(7):817–840, July 2004. 22

[91] Daniel A. Menasce. Web server software architectures. *IEEE Internet Computing*, 7(6), November/December 2003. 25

[92] Daniel A. Menasce, Virgilio A.F. Almeida, and Lawrence W. Dowdy. *Performance by Design: Computer Capacity Planning by Example.* Prentice Hall PTR, 2004. 62, 63

[93] J. Mickens, M. Szummer, and D. Narayanan. Snitch: Interactive decision trees for troubleshooting misconfigurations. In *Proceedings of the 1st Workshop on Tackling Computer Systems Problems with Machine Learning Techniques (SysML)*, April 2007. 33

[94] Microsoft Corporation. Dynamic Systems Initiative. http://www.microsoft.-com/business/dsi/. 3

[95] Microsoft Corporation. DCOM Architecture. http://msdn.microsoft.com/-library/en-us/dndcom/html/msdn_dcomarch.asp. 15

[96] Microsoft Corporation. The .Net Framework. http://msdn.microsoft.com/-netframework/. 15

[97] Microsoft Corporation. WMI - Windows Management Instrumentation. http://www.microsoft.com/whdc/system/pnppwr/wmi/default.mspx. 17

[98] Microsoft Corporation. CLR - The Common Language Runtime. http://msdn.microsoft.com/netframework/programming/clr/default.aspx. 18

[99] Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The paradyn parallel performance measurement tool. *IEEE Computer*, 28(11):37–46, 1995. 35

[100] George A. Milliken and Dallas E. Johnson. *Analysis of Messy Data: Analysis of covariance*. CRC Press, 2002. 75

[101] 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. 17, 33, 35

[102] David Mosberger and Tai Jin. httperf—A Tool for Measuring Web Server Performance. http://www.hpl.hp.com/personal/David63

[103] 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. 49, 152, 162

[104] 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. 49, 152, 162

[105] 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. 162

[106] 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. 162

[107] 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. 162

[108] 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. 162

[109] 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. 29, 162

[110] 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. 162

[111] Mohammad A. Munawar and Paul A.S. Ward. Leveraging many simple statistical models to adaptively monitor software systems. In *International Symposium on Parallel and Distributed Processing with Applications (ISPA)*, 2007. 162

[112] Maitreya Natu and Adarshpal S. Sethi. Efficient probing techniques for fault diagnosis. In *Proceedings of the Second International Conference on Internet Monitoring and Protection (ICIMP)*, page 20, Washington, DC, USA, 2007. IEEE Computer Society. 36

[113] NIST/SEMATECH. Handbook of statistical methods. http://www.itl.nist.gov/div898/handbook/. 75, 76, 81

[114] Object Management Group Inc. CORBA. http://www.corba.org/. 15

[115] Tobias Oetiker. The Round Robin database tool. http://oss.oetiker.ch/-rrdtool/. 34

[116] Soila Pertet, Rajeev Gandhi, and Priya Narasimhan. Fingerpointing correlated failures in replicated systems. In *Proceedings of the USENIX Workshop on Tackling Computer Systems Problems with Machine Learning Techniques (SysML)*, April 2007. 33

[117] 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

[118] Rob Powers, Moises Goldszmidt, and Ira Cohen. Short term performance forecasting in enterprise systems. In *KDD '05: Proceeding of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, pages 801–807, New York, NY, USA, 2005. ACM Press. 26

[119] A. Gonzalez Prieto and R.Stadler. A-gap: An adaptive protocol for continuous network monitoring with accuracy objectives. *IEEE Transactions on Network and Service Management (TNSM)*, 4(1), June 2007. 34, 36

[120] Patrick Reynolds, Charles Killian, Janet L. Wiener, Jeffrey C. Mogul, Mehul A. Shah, and Amin Vahdat. Pip: Detecting the unexpected in distributed systems. In *Proceedings of the Symposium on Networked Systems Design and Implementatio (NSDI)*, pages 115–128, May 2006. 24, 151

[121] Rice University/INRIA. RUBiS - Rice University Bidding System. http://rubis.objectweb.org/. 50

[122] I. Rish, M. Brodie, S. Ma, N. Odintsova, A. Beygelzimer, G. Grabarnik, and K. Hernandez. Adaptive diagnosis in distributed systems. *IEEE Transactions on Neural Networks (special issue on Adaptive Learning Systems in Communication Networks)*, 16(5):1088–1109, September 2005. 36

[123] R. K. Sahoo, A. J. Oliner, I. Rish, M. Gupta, J. E. Moreira, S. Ma, R. Vilalta, and A. Sivasubramaniam. Critical event prediction for proactive management in large-scale computer clusters. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining (KDD)*, pages 426–435, New York, NY, USA, 2003. ACM Press. 26, 30

[124] Abhijit Sawant and Kiran Prabhakara. Admire: An algebraic data mining approach to system performance analysis. *IEEE Transactions on Knowledge and Data Engineering*, 17(7):888–901, 2005. 30

[125] M. Seltzer and C. Small. Self-monitoring and self-adapting operating systems. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VI)*, page 124, 1997. 36

[126] Kai Shen, Ming Zhong, and Chuanpeng Li. I/O system performance debugging using model-driven anomaly characterization. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2005. 25

[127] 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. 17

[128] Christopher Stewart, Terence Kelly, and Alex Zhang. Exploiting nonstationarity for performance prediction. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, pages 31–46, March 2007. 26

[129] Christopher Stewart and Kai Shen. Performance modeling and system management for multi-component online services. In *Proceedings of the Second USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI)*, May 2005. 25

[130] Sun Microsystems Inc. J2EE Management Specification. http://java.sun.-com/j2ee/tools/management/. 12

[131] Sun Microsystems Inc. Java Management Extensions (JMX) Technology. http://java.sun.com/javase/technologies/core/mntr-mgmt/-javamanagement/. 13, 19, 22

[132] Sun Microsystems Inc. The Java Virtual Machine Specification. http://java.-sun.com/docs/books/vmspec/. 18

[133] Sun Microsystems Inc. The JVM Tool Interface. http://java.sun.com/j2se/-1.5.0/docs/guide/jvmti/. 18

[134] Sun Microsystems Inc. Platform Monitoring and Management Using JMX. http://java.sun.com/j2se/1.5.0/docs/guide/management/agent.html. 18

[135] Sun Microsystems Inc. HPROF: A Heap/CPU Profiling Tool in J2SE 5.0. http://java.sun.com/developer/technicalArticles/Programming/-HPROF.html. 18

[136] Sun Microsystems Inc. J2EE 1.4 Platform Specification. http://java.sun.com/j2ee/j2ee-1_4-fr-spec.pdf. 15

[137] Symantec Corp. *Symantec Indepth for J2EE 8.0 - User's Guide*. Number 287103. 2007. http://support.veritas.com/docs/287103. 35

[138] Ariel Tamches and Barton P. Miller. Fine-grained dynamic instrumentation of commodity operating system kernels. In *Proceedings of the third Symposium on Operating Systems Design and Implementation*, February 1999. 17

[139] 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. 2

[140] Transaction Processing Performance Council. TPC-W – a transactional web e-Commerce benchmark. http://www.tpc.org/tpcw/. 50

[141] Bhuvan Urgaonkar, Giovanni Pacifici, Prashant Shenoy, Mike Spreitzer, and Asser Tantawi. An analytical model for multi-tier internet services and its applications. In *Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 291–302, New York, NY, USA, 2005. ACM Press. 25

[142] Mustafa Uysal, Tahsin M. Kurc, Alan Sussman, and Joel H. Saltz. A performance prediction framework for data intensive applications on large scale parallel machines. In *Proceedings of the Fourth Workshop on Languages, Compilers and Runtime Systems for Scalable Computers*, pages 243–258, 1998. 25

[143] Robbert van Renesse, Kenneth P. Birman, and Werner Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Transactions on Computing Systems*, 21(2):164–206, 2003. 22, 34

[144] Helen J. Wang, John C. Platt, Yu Chen, Ruyun Zhang, and Yi-Min Wang. Automatic misconfiguration troubleshooting with peerpressure. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, pages 17–17, 2004. 33

[145] Sanford Weisberg. *Applied Linear Regression.* Wiley-Interscience, 3rd edition, 2005. 75

[146] Ian H. Witten and Eibe Frank. *Data Mining: Practical machine learning tools and techniques.* Morgan Kaufmann, San Francisco, 2nd edition, 2005. 53

[147] Jeffrey M. Wooldridge. *Introductory Econometrics: A Modern Approach*. South-Western Educational Publishing, 1st edition, 2000. 75

[148] 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. 32

[149] Q. Zhang, L. Cherkasova, and E. Smirni. A regression-based analytic model for dynamic resource provisioning of multi-tier applications. In *Proceedings of the International Conference on Autonomic Computing (ICAC)*, 2007. 26

[150] Steve Zhang, Ira Cohen, Moises Goldszmidt, Julie Symons, and Armando Fox. Ensembles of models for automated diagnosis of system performance problems. In *Proc. of the International Conference on Dependable Systems and Networks (DSN'05)*, pages 644–653, 2005. 31

[151] Tao Zheng, Jinmei Yang, Murray Woodside, Marin Litoiu, and Gabriel Iszlai. Tracking time-varying parameters in software systems with extended kalman filters. In *CASCON '05: Proceedings of the 2005 conference of the Centre for Advanced Studies on Collaborative research*, pages 334–345. IBM Press, 2005. 26