SEMANTIC-AWARE ANOMALY DETECTION IN DISTRIBUTED STORAGE SYSTEMS.

by

Saeed Ghanbari

A thesis submitted in conformity with the requirements for the degree of Doctor of Philosophy
Graduate Department of Electrical and Computer Engineering
University of Toronto

© Copyright 2014 by Saeed Ghanbari
Abstract

Semantic-aware Anomaly Detection in Distributed Storage Systems.

Saeed Ghanbari
Doctor of Philosophy
Graduate Department of Electrical and Computer Engineering
University of Toronto
2014

As distributed storage systems become central to business operations, increasing their reliability becomes an essential requirement. Towards this goal, distributed storage systems come equipped with various monitoring probes. As a result, they generate massive amounts of monitoring data. The amount and complexity of such monitoring data surpasses the human ability to detect anomalies in this data. Our thesis is that, in order to obtain a practical solution to the reliability issues posed by modern distributed storage systems, the semantics of these systems must be incorporated into the anomaly detection process. Our semantic-aware approach enables digesting large amounts of monitoring data, through statistical methods, into a few human understandable patterns. These selected unusual patterns provide intuitive pinpointing towards the possible root causes of an anomaly; they can be leveraged by the human in the loop for root-cause anomaly analysis and related investigations. Our strengths lie in the lightweight, minimal overhead of our techniques, as well as their sensitivity to detect even subtle and localized unusual patterns.

In this dissertation, we introduce three semantic-aware anomaly detection techniques: i) anomaly detection based on validation of high-level invariants, ii) stage-aware anomaly detection through runtime tracking of execution flows inferred from existing log statements in the code, and iii) a composite anomaly detection model using cross-component correlations.

We studied our methods in two realistic environments: networked multi-tier storage systems used in enterprise settings, and a variety of scalable distributed storage platforms used in large data centers, i.e., HBase, HDFS and Cassandra. Our evaluation shows that our methods are lightweight, introducing negligible overhead, and detect anomalies in real-time.
Dedication

To the love of my life,

Maryam.
Acknowledgements

This journey was not possible without help and contributions of many people. My thanks goes to my supervisor, Cristiana Amza, for her support, encouragement and guidance during all these years, and to my supervisory committee Michael Stumm and David Lie for their invaluable feedback. I thank my friends at University of Toronto, Madalin Mihailescu, Gokul Soundararajan, Jin Chen, Mihai Burcea, Reza Azimi, Livio Soares, Ioana Baldini, Weiihan Wang, Adrian Popescu, Daniel Lupei, Afshar Ganjali, and Reza Nakhjavani. My special gratitude goes to Ali Baradaran Hashemi whose help made the completion of this thesis possible.
# Contents

1 Introduction ........................................... 1
   1.1 Contributions ................................... 4
   1.2 Outline .......................................... 5

2 Background ........................................... 7
   2.1 Reliability Principles and Challenges in Modern Systems ............... 7
   2.2 Sources of Monitoring Data in Production Systems ......................... 8
      2.2.1 Performance metrics .................................. 8
      2.2.2 Operational logs .................................... 9
   2.3 Causes of Failure .................................... 10
   2.4 Anomaly Detection and Root-cause Analysis ................................ 10
   2.5 Data Mining Approaches to Anomaly Detection ............................. 11
      2.5.1 Mining Performance Metrics ............................. 11
      2.5.2 Mining Operational Logs ............................... 11

3 Anomaly Detection based on Invariant Validation .......................... 13
   3.1 Introduction ......................................... 13
      3.1.1 Language Constructs and Semantically-tagged Models ............... 14
   3.2 Architectural Overview .................................. 15
      3.2.1 The SelfTalk Language ................................ 16
      3.2.2 The Dena Runtime System .............................. 17
   3.3 The SelfTalk Language .................................. 18
      3.3.1 Hypothesis ......................................... 19
      3.3.2 Query ............................................ 22
   3.4 Validating Expectations .................................. 24
4.5.2 Experimental Setup .................................................. 58
4.5.3 Overhead ................................................................. 59
  Overhead of Task Execution Tracking ................................ 59
  Storage Overhead .......................................................... 60
  Statistical Analyzer Overhead ........................................... 60
4.5.4 Cassandra ............................................................... 61
  Usecase: Uncovering Masked Failure .................................. 66
4.5.5 HBase/HDFS ............................................................ 67
  Fault Diagnosis ............................................................. 68
  Usecase: Uncovering Bugs and Misconfigurations ................. 71
4.5.6 False Positive Analysis ............................................... 71
4.5.7 Summary of Results .................................................. 72
4.6 Discussion ............................................................... 73
4.7 Conclusion ............................................................... 75

5 Composite Anomaly Detection using Cross-component Correlations 77
  5.1 Introduction ............................................................ 77
  5.2 Background and Motivation ........................................... 79
    5.2.1 Motivating Example ............................................... 79
    5.2.2 Discussion of Results and Goals ................................ 80
  5.3 Semantic-Driven System Modeling Using a Belief Network .... 81
    5.3.1 Overview .......................................................... 82
    5.3.2 Operation .......................................................... 83
    5.3.3 Expert Knowledge Specification ................................ 83
    5.3.4 Querying the Network for Anomaly Diagnosis ............... 84
    5.3.5 Belief Network Meta-Model with Bayesian Learning ........ 84
      Encoding Tiers, Components and Inputs .......................... 85
      Encoding Dependencies between Variables ......................... 85
      Training Using Fault Injection ..................................... 85
      Inferences within the Belief Network ............................... 86
    5.3.6 Discussion of Trade-Offs ........................................ 86
  5.4 Prototype and Testbed ............................................... 86
5.5 Experiments ......................................................... 88
5.6 Summary ........................................................... 91

6 Related Work ......................................................... 93
   6.1 Black-box Statistical Approaches ............................... 93
   6.2 Theoretical Model-Based Approaches ......................... 94
   6.3 Expectation-Based Approaches ................................. 94
   6.4 Mining Operational Logs ........................................ 95
   6.5 Inferring Execution-Flow ......................................... 96
   6.6 Inferring Causality .............................................. 97

7 Conclusion and Future Work ......................................... 98
   7.1 Summary .......................................................... 98
   7.2 Future Work ..................................................... 99
      7.2.1 SelfTalk and Dena ........................................ 99
      7.2.2 SAAD Analyzer ........................................... 100
      7.2.3 Composite Anomaly Detector using Cross-Component Correlations ............................ 100

Bibliography ............................................................. 101
Chapter 1

Introduction

Storage systems have become an indispensable part of modern life. In the past decade, the volume of generated data has exponentially increased as a result of proliferation of social media and mobile computing, and in general, due to the pervasiveness of automated environments supporting various aspects of human life. For most modern businesses, storing and retrieving data efficiently is now on the critical path of their operation, which increases the importance of providing dependability and reliability of storage systems. Despite rigorous tests performed during the development cycle to minimize chances of failure, unpredictable situations in the field cause storage systems to malfunction or even crash [50].

Failures in a production system affect productivity and profitability of a business. The damage incurred by any downtime may cost a business substantial revenue loss, and can even push it to the brink of bankruptcy. Therefore, it is crucial to diagnose and repair failures quickly.

A storage system is typically equipped with various monitoring probes, such as performance metrics and console logs, that allow users to constantly monitor the health state of individual components. Users of production systems search for anomalous patterns in the monitoring data and use these patterns as pointers to the root cause of an anomaly. However, finding relevant anomalous patterns in the monitoring data of a production system is still challenging.

Typically, failures are detected by low-level alerts generated by system components, e.g., error logs and performance metric thresholds. While the low-level alerts inform users instantly of a failure, alerts typically appear in multiple places, due to dependencies between various system components. The diversity of sources of monitoring data, and the sheer volume of data collected from many components, and at multiple layers of software stacks, pose challenges for anomaly diagnosis. Too many alerts or a large volume of data to analyse can overwhelm users and can make it hard to pinpoint the faulty component(s).
For instance, a cluster of 100 servers, with a standard installation of HBase, a distributed columnar key/value store that empowers major social network websites including Facebook, running on top of Hadoop Distributed File System, a distributed fault tolerant file system, exposes 30000 different metrics through its default monitoring infrastructure. The above mentioned cluster generates monitoring data every 15 seconds under light load; a total of 172 million data points are generated per day. The growth in the volume of monitoring data is especially a challenge for operational logs. The above mentioned cluster generates about 24000 log messages per minute, or 400GB of log data per day.

Besides the overwhelming diversity and volume of data, the format in which this data is presented to users is often difficult to interpret. Monitoring data is typically in the form of sparse log messages and/or aggregate values, such as average performance metrics. As a consequence, diagnostic data in a production system lacks both high-level and in-depth meaning. Moreover, monitoring data may be ambiguous or incomplete, depending on the performance impact versus level of detail trade-off chosen by the user.

Specifically, although monitoring is an essential ingredient of a reliable production system which usually has monitoring probes active at all times, these monitoring probes are second-class citizens: they must not interfere with the primary task of the system. Unlike debugging and quality assurance that come at the (generally accepted) expense of high performance overhead, monitoring probes must have a minimal footprint on the system performance and associated system resources.

As a result, monitoring data collected from production systems are typically limited in scope and amount of diagnostic information that they can provide. For instance, the logging level in a production system is often set to event logging, which generates log messages only for major events, such as error messages. These log messages are sparse and do not provide insight into the execution flow leading to the error.

Various automated anomaly detection solutions, based on statistical machine learning, have been proposed. Most of these methods require high-resolution monitoring data through execution tracing \cite{27,88,38} or verbose logging \cite{109,110,74}. Since capturing high resolution monitoring data imposes undue overhead on the system, these methods are unsuitable for production environments that require constant, always-on, low-overhead monitoring.

Other existing solutions use blackbox, standard data-mining methods to find anomalous patterns \cite{40,29}. These methods are agnostic to the semantics of the system. Although they detect anomalies in production monitoring data, the outcome of these methods is presented to the user as a set of anomalous patterns lacking a meaningful connection between the anomalous pattern and the high-level semantics of the system. In the absence of such connections, users find it challenging to interpret the monitoring information, and lack guidance towards finding the root-cause of an anomaly.

In summary, with state of the art solutions to the problem of anomaly detection, users have two options:
i) pay the high performance and resource costs of detailed monitoring, which is not an option in a production environments, or ii) use sparse monitoring in production systems in conjunction with black-box data analysis solutions that leave users with hard-to-interpret results.

In this dissertation, we leverage the semantics of the system to detect performance anomalies in production storage systems from sparse low-cost monitoring data.

Our thesis is that, by augmenting monitoring data with a high level context, pertaining to execution flow and/or the functional/structural properties of the system, we can interpret anomalies with reference to a semantic model that provides both in-depth and meaningful information to the user. In this context, our claim is two fold: i) that we can develop techniques that provide these semantic associations at a very low cost, and ii) that these techniques, in conjunction with standard statistical analysis methods, can provide effective automated support for anomaly detection.

Associating the human understandable semantics of the system to the process of anomaly detection has several benefits.

First, the detected anomalies are readily interpretable, because they are expressed as the violation of an expected behavior. Second, searching for anomalies based on the semantics of the system e.g., for violations to known invariants, may compensate for any lack of high-resolution monitoring data in the production environment. Conversely, in the presence of high resolution monitoring data, semantic-awareness combined with statistical analysis restricts the anomaly detection search space. Both user and machine need to investigate only patterns that are unusual and also relevant to the semantics of the system. As a result, statistical models guided by semantics detect anomalies efficiently with minimal computational and storage resources.

The techniques for supporting users in their anomaly detection tasks that we introduce in this dissertation rely on existing and widely available monitoring sources, i.e., performance metrics and log statements. Our proposed methods require no or minimal modifications to existing systems, and incur near-zero runtime overhead.

We introduce three anomaly detection tools based on the following semantics:

1. **Invariants.** We find anomalies that violate the high-level expected invariants which must hold in a healthy system. These invariants are high-level expected mathematical properties that must hold over the performance metrics and configuration settings of individual components and/or multiple components. These mathematical properties are either based on known operational laws, such as queuing network theory, and/or the design objectives of the component; e.g., given a cache component, one would expect that I/O requests that are served from a cache module have lower latency than those that miss in the cache. Users can reason about the violation of these expected behaviors to limit the anomaly detection search to those patterns that are descriptive of the root-cause of problems.
2. **Staged Architecture.** We leverage the stage-oriented architecture of high-performance storage servers, where the complex logic of a server is broken down into a set of smaller code modules, called stages. The execution flows within each stage are expected to show statistical similarity. We leverage the existing log statements in the code to track task execution flows within each stage. Incorporating stage names and execution flow semantics as the high level context limits the search space for root causes by pinpointing specific anomalous code stages, and reduces compute and storage requirements for log analysis.

3. **Structural Dependencies.** Finally, we encode the dependency between tiers in a multi-tier system. Due to dependencies between tiers, a fault in one tier may induce anomalies in other tiers. To limit the scope of anomaly detection search, we encode the structural dependencies across tiers in a semantic network. The semantic network accurately pinpoints the most likely faulty tier, by ignoring the anomalies in the dependent tiers.

In the following, we describe in more detail each of the above semantic-aware anomaly detection and diagnosis techniques.

### 1.1 Contributions

In this dissertation, we implemented three novel anomaly detection and diagnosis tools based on semantic-awareness, and we validated our tools on two main stream storage systems: i) a multi-tier networked storage system that resembles the storage systems used in enterprise environments, and ii) a set of scalable distributed storage systems which are used in web-scale data-intensive applications, such as social networks.

- **Performance Anomaly Detection based on Invariant Validation:** As modern storage systems are becoming increasingly large and complex, it becomes more difficult for users to understand the overall behavior of the systems, and diagnose performance problems. To assist the inspection of performance behavior of the system, we introduce *SelfTalk*, a novel declarative language that allows users to query and understand the status of a large scale storage system. *SelfTalk* is sufficiently expressive to encode a user’s high-level hypotheses about system invariants, normal correlations between system metrics, or other a-priori derived performance models. Given a hypothesis expressed in *SelfTalk*, such as, “I expect that the throughputs of interconnected system components are linearly correlated”, *Dena*, our runtime support system, instantiates and validates it using actual monitoring data within specific system configurations. We evaluate *SelfTalk/Dena* by posing several hypotheses about system behavior and querying *Dena* to validate system behavior in a networked storage server. We find that *Dena* automatically validates the system performance based on the pre-existing hypotheses and helps to diagnose system misbehavior. Specifically, we validate
the behavior of caches, request schedulers, and the interplay between the different components. *Dena* validates the user’s expectation within 2 seconds on over 10GB of monitoring data [53].

- **Stage-Aware Anomaly Detection by Runtime Tracking of Execution Flows from Log Statements:**
  
  We introduce Stage-aware Anomaly Detection (SAAD), a low-overhead real-time solution for detecting runtime anomalies in storage systems. Modern storage server architectures are stage-oriented and multi-threaded. SAAD leverages this to collect stage-level log summaries at runtime and to perform statistical analysis across stage instances. Stages that generate rare execution flows and/or register unusually high duration for regular flows at run-time indicate anomalies. SAAD makes two key contributions: i) limits the search space for root causes, by pinpointing specific anomalous code stages, and ii) reduces compute and storage requirements for log analysis, while preserving accuracy, through a novel technique based on log summarization. We evaluate SAAD on three distributed storage systems: HBase, Hadoop Distributed File System (HDFS), and Cassandra. We show that, with practically zero overhead, we uncover various anomalies in real-time.

- **Composite Anomaly Detection in Multi-tier Systems through Leveraging Dependencies:** We introduce a semantic-driven approach to system modeling for improving the accuracy of anomaly diagnosis. Our framework composes heterogeneous families of models, including generic statistical models, and resource-specific models into a belief network, i.e., Bayesian network. Given a set of models which sense the behavior of various system components, the key idea is to incorporate expert knowledge about the system structure and dependencies within this structure. Expert beliefs about the system hierarchy, relationships and known problems can guide learning, but do not need to be fully specified. The system dynamically evolves its beliefs about anomalies over time. We evaluate our prototype implementation on a dynamic content site running the TPC-W industry-standard e-commerce benchmark. We sketch a system structure and train our belief network using automatic fault injection. We demonstrate that our technique provides accurate problem diagnosis in cases of single and multiple faults. We also show that our semantic-driven modeling approach effectively finds the component containing the root cause of injected faults, and avoids false alarms for normal changes in environment or workload [52].

### 1.2 Outline

The outline of this dissertation is as follows: Chapter 2 provides the necessary background on the challenges and limitations of anomaly detection at large-scale systems and discusses previous work addressing the issues discussed in the subsequent chapters. Chapter 3 presents a query language and runtime system to assist the users
in inspecting performance behavior according to his/her expert knowledge. Chapter 4 presents a realtime anomaly detection technique which leverages semantics of the code structure and log statements of high-performance storage systems. In Chapter 5 we present a framework to semantically combine multiple anomaly detection methods for root-cause analysis. Chapter 6 discusses the related work. Chapter 7 concludes the dissertation and outlines future work.
Chapter 2

Background

In this chapter, we provide the background for the techniques presented in this dissertation. We introduce the reliability challenges in modern systems. We next describe the usual sources of monitoring data and the typical causes of failures in modern distributed systems with a brief coverage of previous work in addressing the challenges of anomaly detection in such systems.

2.1 Reliability Principles and Challenges in Modern Systems

Computer systems have become an indispensable part of modern life. In the past decade, human society has been fundamentally transformed by the ubiquity and scale of computing systems. Exponential increase of computing power combined with dramatic advancement in telecommunication technology have enabled businesses to offer a wide range of services to a virtually unlimited number of users. As computer systems become central to business operations, their dependability and reliability become critical.

In his seminal work, Jim Gray argues that the key to building reliable systems is to make them fault-tolerant, through following two core design principles in their architecture: modularity and redundancy [54]. Modularity prevents the failure of a module from affecting the availability of the whole system, while redundancy enables fast fail-over. Today, modern distributed systems adopt this vision and the associated core design principles towards providing reliability for systems composed of otherwise unreliable computing components.

Despite having a design built on sound modularity and redundancy principles, modern systems still register crashes to such a degree that these crashes are sometimes reported in the media [5, 6]. The reasons for this are two fold.

First, fault tolerance by modularity and redundancy is based on the assumption that when a module fails, it stops immediately (fail-stop) and other modules immediately detect that it is stopped (fail-fast). These two
properties are essential for an immediate fail-over to avoid performance degradation of the whole system. But in many cases, failure in a module manifests as a gradual performance degradation. Detecting these failures is not trivial because of the fuzzy nature of the performance degradation. Due to dependency between modules, the performance degradation of the failing module may cascade to other modules, and eventually hurt the performance of the whole system.

Second, fault tolerance through fail-over may mask serious failures in the long run that can eventually lead to catastrophic outcomes. In systems where a fault is masked by means of fail-over, it is not trivial to discern a transient fault from a serious fault that jeopardizes the whole system. Hence, a serious fault may remain undetected until it affects the whole system.

For these reasons, designing methods to detect and diagnose failures has proved difficult [71, 37, 38, 41, 107, 22, 90].

While this is a hard, inherent problem, which we do not claim to completely solve with our work in this dissertation, our methods attempt to address and alleviate it in two ways: i) by relying on statistical techniques that learn to differentiate between usual and unusual performance patterns of the system over time and ii) by associating semantic information with monitoring data to help users understand what happened at the time of performance degradation, e.g., to distinguish normal fail-over cases from cascading or terminal faults.

### 2.2 Sources of Monitoring Data in Production Systems

Without high quality monitoring data, operating a large-scale production system is almost impossible in modern environments. Collecting, transferring, and analyzing monitoring data is not free. These tasks consume computing and storage resources. A salient desired characteristic of system monitoring in a production environment is to be low overhead such that monitoring does not interfere with the primary workload.

The sources of monitoring data in production systems can be categorized into two groups: i) performance metrics, and ii) operational logs.

#### 2.2.1 Performance metrics

Performance metrics are numerical values measuring some aspect of a system, e.g., CPU utilization and number of RPC calls per second. Most operating systems expose their resource accounting information. Server applications, such as databases, web servers, and storage systems, as well as devices, such as, network switches and routers, report hundreds of performance metrics. Performance metrics are suitable for graphical representation through charts and trend lines. In industry, there are many technologies that provide full solutions for collecting and presenting performance data in distributed systems, for example, Ganglia [4], Nagios-Collectd [9], openTSDB [11],...
To keep the overhead of collecting performance metrics down, performance metrics measure the aggregate behavior of system components. As a result they do not provide high-resolution in-depth insight into the internals of the system. For example, CPU utilization measures the CPU busy time versus the idle time in a time period, without providing details of what portion of the application is CPU intensive.

In contrast to the production environments, users have access to deep monitoring data of programs through tracing during development and testing cycles. Tracing/profiling techniques insert a set of probes to collect deep monitoring data from a running application through dynamic and/or static instrumentation. The probes often incur high overhead, but provide a precise view of execution of a program. Dtrace [32], Xtrace [49], Systemtap [17], Oprofile [12], Stardust [103], Fay [48], Dapper [93], are examples of tracing/profiling tools. These tools are used in production systems in a limited way. To curb the overhead of overall system performance, tracing/profiling is activated in short bursts or invoked periodically.

### 2.2.2 Operational logs

Operational logs capture high resolution information about a running system, such as the internal execution flow of all individual requests and the contribution of each to the overall performance. Insights gained from them are critical diagnostic data in production systems.

Modern logging libraries allow users to configure the level of logging desired. At an INFO-level logging (also referred to as non-verbose logging and event logging), log messages register major events in the system, such as establishing a session or reporting an error. With DEBUG-level logging, (also referred to as verbose logging and trace-level logging), log messages provide detailed, high resolution information about execution flow.

Because DEBUG-level logging provides a fine-grained view of the program execution, it generates substantially more log messages than INFO-level logging. For instance, a 10-node Cassandra cluster, a distributed peer-to-peer columnar key/value store, generates 500000 log messages per hour (5TB log data per day) when configured to DEBUG-level logging, under a moderate workload. This number is about 2600 times more than the number of log messages at INFO-level logging. Due to overhead concerns, logging is often set to INFO-level logging in production environments.

The volume of log data grows fast. Production systems usually have a dedicated sub-system to collect, transfer and manage the operational logs: Splunk [15], Hadoop Chukwa [31], LinkedIn Kafka [69], Facebook Scribe [104] and Cloudera Flume [1].
2.3 Causes of Failure

The underlying causes for performance failure are diverse. There is little consensus on how to define a single root cause. For instance, consider a web server that is slow as a result of too many open sockets. The root cause in this case can be argued to be a bad design decision that led to an inefficient use of open sockets. It can also be argued that this is a misconfiguration that set a low limit on the allowed number of open files. Or, it can be argued that an excessive load on the web server is to blame for the failure. This simple example illustrates that the root-cause is subject to diverse interpretations.

Nevertheless, there have been isolated attempts to characterize the causes of failures more formally. Although some studies indicate that the number of failures with undetermined root cause is significant [91], a recent study on NetApp customer logs reveals that three major sources of failure in enterprise production systems are misconfiguration, software defects, and hardware failure [64]. The study indicates that hardware failures occur more frequently than other types of failure, although they are easy to detect. The study points out that the diagnosis of software bugs and misconfigurations is the main challenge. This finding is consistent with earlier observations [55, 84]. Software bugs account for a small portion of failure causes, but they require long resolution times.

2.4 Anomaly Detection and Root-cause Analysis

The goal of anomaly detection is to find the root cause of a failure. As discussed in the previous sections, the causes of failures are diverse and are subject to human interpretation. In fact, precise diagnosis of any failure requires human intelligence. Therefore, finding the root-cause of a problem depends on accurately detecting the anomalous patterns involved. By analogy, in the medical field, the ability of a physician to correctly diagnose the cause of an illness depends on finding the symptoms of the disease which appear as some form of anomalous patterns. Similarly, in large distributed systems, anomaly detection uncovers anomalous patterns, which help users diagnose the root cause of problems.

Anomaly detection helps the user detect and pinpoint patterns that do not conform to an established normal behaviour [34]. A behaviour is considered normal if it is consistent with an expected behaviour. For example, a normal behavior of a cache is that by increasing the cache size, the miss ratio decreases or at least stays constant under the same workload. As we will discuss in the Chapter 3, defining and validating these expectations on a large volume of monitoring data is not trivial. We introduce Dena and SelfTalk that enable users to define and validate these expectations, as a means to detect anomalies.

The other definition of normal behaviour is based on a dominant behaviour over time. A pattern that is different from the patterns previously seen in the historical operation of the system is considered anomalous. These
patterns are often called outliers, and methods that detect this type of anomalies are called “outlier detection” methods. Outlier detection methods are based on statistical concepts. An outlier is defined as a sample that deviates significantly, according to a statistical index, from other members of the sample set in which it is observed.

A challenge that the outlier detection method must address is to distinguish an outlier that is caused by an anomaly (true positive) from an outlier that is caused by imprecision and noise in data (false positive).

2.5 Data Mining Approaches to Anomaly Detection

Data mining is the computational process of discovering patterns in large data sets. Data mining techniques are a mixture of methods from the areas of artificial intelligence, machine learning, statistics, and text processing. The overall goal of the data mining process is to extract information from a data set and transform it into an understandable structure for further use [33]. With the dramatic increase in the volume and complexity of monitoring data in the past decade, data mining techniques have become a viable approach to anomaly detection.

2.5.1 Mining Performance Metrics

In principle, methods that mine performance metrics for anomaly detection are based on detecting violations in the statistical correlations between metrics [40, 41, 76, 28, 29]. For instance, Cohen et al. [41] use a restricted Bayesian classification method to correlate service level violations to the performance metrics. Guofei et al. [63] leverage pairwise correlations between variations of the performance metrics and detect anomalies in terms of violations of the learned correlations.

These methods are agnostic to the semantics of the system. The results of these techniques often lack the association between detected anomalies and the high-level semantics of the system.

2.5.2 Mining Operational Logs

Unlike performance metrics, operational logs contain information of execution flow and sequence of events in a running application. The common denominator of the log mining techniques is to i) link log messages to the log statements in the source code, ii) group related log messages, e.g., per individual tasks, to infer execution flow, and iii) detect anomalies in the executed tasks.

To identify log statements corresponding to the log messages, the log data is processed using regular expression matching. Since the volume of log messages is on the order of Gigabytes or even Terabytes, regular expression matching is often parallelized on data analytics infrastructures, such as MapReduce, to speed up the process.
Grouping related log messages is commonly done through manual, ad hoc rules [80]. For instance, log messages that belong to an execution flow might not appear in a contiguous order due to thread interleaving. To address this problem, users may add thread identifiers to the log messages and then define parsing rules to identify thread ids from log messages to group them per thread. This will solve inferring execution flow per thread.

However, a thread might be reused to execute several unrelated tasks during its lifetime. Hence, distinguishing the execution flow of individual tasks from the log messages that have the same thread id is challenging. For this reason, existing log analytics solutions require users to manually define a set of rules to group the log messages that belong to a task. In more detail, these rules denote certain log messages as markers of start and end of individual tasks. Then, the users can group all log messages pertaining to the same thread that appear in between these two markers as a task. In a server that consists of thousands of log statements with virtually infinite combinations of log messages that the system would possibly generate, e.g., because of multiple control flows and task exit points, defining rules to identify these markers is challenging.

This makes ad-hoc methods to group log messages [51, 75, 101, 109] cumbersome and inexact.

In Chapter 4, we leverage the code structure of servers to systematically identify individual tasks at runtime and infer their execution flows from existing log statements in the code.
Chapter 3

Anomaly Detection based on Invariant Validation

In this chapter we introduce a set of tools for aiding human administrators with detecting and diagnosing functional and performance anomalies in generic multi-tier distributed storage systems.

3.1 Introduction

As modern multi-tier systems increase in scale and complexity, and their applications become more sophisticated, performance validation and diagnosis of these systems becomes more and more challenging. Many commercial tools for monitoring and control of large-scale systems exist; tools such as HP’s Openview and IBM’s Tivoli products collect and aggregate information from a variety of sources and present this information graphically to users. However, the complexity of the information available with such tools exceeds the human ability to use these diagnosis tools effectively [67].

In this chapter, we study high-level paradigms and advanced tools for performance analysis, and interactive performance diagnosis of multi-tier distributed storage systems.

Our semantic-aware approach to reliability in generic distributed storage systems allows users to instruct the system, in a high level language, to validate a set of given invariants that reflect user beliefs about normal system operation.

We argue that the system architect, the performance analyst, and/or the system administrator have a wealth of information and expertise about the system, which can be expressed as performance hypotheses. Performance hypotheses approximate system performance metrics, as a function of other system metrics, the system structure
and/or system configurations. The system architect or analyst can provide pre-built performance hypotheses, in the form of statistical correlations, models, invariants, other mathematical laws or properties describing normal system behavior for distribution to site operators. For example, performance hypotheses can be derived based on a variety of existing automated or semi-automated performance modeling techniques, such as automatic statistical correlations across monitored metrics [39, 62], black-box performance model generation by interpolation from experimental samples [95, 96], analytical system modelling [97], or gray-box approaches, where high-level semantic knowledge of the system is used to guide experimental sampling [97].

Our tools provide the human with a bird’s eye view of the performance and reliability status of the infrastructure in terms of validating a set of invariants within actual semantic contexts. The system monitors its own patterns and infers through statistical analysis which patterns are worth further investigation by the human due to a violation of monitored invariants. Our tools selectively present these patterns at a high level in a semantically meaningful way.

3.1.1 Language Constructs and Semantically-tagged Models

Regardless of the method for building hypotheses, they are validated based on monitoring data and stored in a knowledge base. System administrators can then check the accuracy of existing performance hypotheses in the field, or even design hypotheses specific to their local configurations. The system analyst or administrator can also use stored hypotheses to diagnose the behavior of the system periodically with newer monitoring data. For the remainder of the chapter, we refer to the various persons that provide, monitor, and check hypotheses as analysts.

In order to assist analysts with the tasks of expressing, validating, and refining their hypotheses about the system, as well as diagnosing system problems interactively, we introduce a novel validation infrastructure: (i) a new high-level declarative language, called SelfTalk, for expressing and refining hypotheses about the system and (ii) new runtime support, called Dena, capable of self-monitoring system metrics, evolving, and adapting dynamic models of metric correlations, as well as interacting with its administrator in SelfTalk.

SelfTalk allows users to express their understanding of the semantics of the system in a declarative query language. Dena is a runtime engine, which builds semantically-tagged models based on the hypotheses provided by the user and actual monitoring data on the fly. Dena thus also verifies the given user beliefs against monitoring data. If a previously built model for a system invariant no longer holds, this may indicate an anomaly in one or more system components.

The relationship between metric classes within a hypothesis can be expressed in SelfTalk as a high-level, human-readable expression, such as, “linear”, “less than”, “equal to”, or “monotonically decreasing”, e.g., “I expect that the average query latency measured at the database system is greater than the average data access la-
tency measured at its back-end storage server." Our language can also leverage existing models, or encode generic laws that govern system behavior, such as Little’s law \textsuperscript{[106]} that correlates throughput and latency values, or the monotonically decreasing property of the miss-ratio curve (MRC) \textsuperscript{[115]} in a system cache. Each performance hypothesis expressed in \textit{SelfTalk} contains as qualifiers the contexts where the hypothesis is expected to apply, i.e., the set of configurations, or workloads where the pre-computed models or metric correlations expressed in the hypothesis hold.

By submitting a \textit{SelfTalk} query, the analyst can validate or refine, existing hypotheses, or add new ones interactively as the system configurations and the application set evolve. For this purpose, \textit{Dena} parses each hypothesis, selects the appropriate mathematical expression that corresponds to the given keywords, and instantiates the hypothesis into a set of concrete \textit{internal expectations}, one for each matching system configuration. Then, \textit{Dena} evaluates compliance for the set of expectations by fitting the monitoring data within any given context to the mathematical expression. \textit{Dena} enters the hypothesis, matching expectations, contexts, and a confidence score into the knowledge base. In addition, \textit{Dena} can be configured to validate hypotheses periodically and generate alarms if previously derived correlations do not fit current monitoring data. In case of alarms, \textit{Dena} pinpoints the hypotheses that became invalid and the components affected, thus giving valuable feedback to the analyst.

We perform an evaluation of our approach by posing several hypotheses to understand normal behavior of the overall system, the behavior of individual components, and to diagnose misbehavior in a multi-tier dynamic content server consisting of a Apache/PHP web server, a MySQL database using virtual volumes hosted on a virtual storage system called \textit{Akash} \textsuperscript{[97]} running industry-standard benchmarks: TPC-W and TPC-C. We find that \textit{Dena} can quickly validate system performance to analysts’ hypotheses and can help in diagnosing faults, or other system misbehavior. In addition, we use \textit{Dena} to validate different components, such as the MySQL cache, the storage system I/O scheduler, and the behavior of multi-level caches.

The rest of the chapter is organized as follows. Section \textit{3.2} presents the \textit{SelfTalk} language and an overview of the overall system. We expand on the language in Section \textit{3.3} and the design of \textit{Dena} in Section \textit{3.4}. Section \textit{3.5} describes our prototype and our experimental setup consisting of a multi-tier system consisting of MySQL databases using virtual storage on the \textit{Akash} storage server. Section \textit{3.6} presents results of our experiments showing how \textit{Dena} allows the analysts to probe and understand the overall behavior and the per-component behavior in a multi-tier system. Section \textit{3.7} summarizes the chapter.

### 3.2 Architectural Overview

We introduce a novel language and runtime support for understanding and validating the behavior of a multi-tier dynamic content server system. Specifically, we design a high-level declarative language, called \textit{SelfTalk},
which allows an analyst to express generic hypotheses about normal system behavior, including operational laws and relationships between metric classes. The analyst submits hypotheses in SelfTalk to a runtime system called Dena, which is in charge of instantiating and validating them, based on automatic metric monitoring, statistics collection, and correlation at various points in the multi-tier system. In the following, we describe our language, the design of Dena, our tool, and how the analyst and the system interact to check compliance to expectations.

### 3.2.1 The SelfTalk Language

A hypothesis consists of a relationship on a set of metric classes and the associated validity context for that relationship. The context can be a set of configurations or workload properties that could potentially affect the given relationship. If the relationship is believed to be an invariant, then its corresponding context is empty. We provide some examples of hypotheses written in SelfTalk; these highlight the simplicity of the language and its ease of use. A simple invariant that can be checked by the analyst is that the number of cache misses (num_cache_miss) must be less than or equal to the number of cache accesses (num_cache_gets), as shown in Listing 3.1. This is a simple hypothesis issued by the analyst trying to understand the behavior of a cache in a multi-tier system; she does not have to know the details of the cache such as its replacement policy and only needs to have high-level understanding. She simply states that for a given cache, she expects the number of cache misses to be less than the number of cache accesses. This is an invariant of the cache – that is, it must hold true for all configurations and workloads. Thus, the analyst can submit the hypothesis without a context and Dena will check if this relationship is indeed valid for all configurations.

However, some hypotheses are valid only for particular configurations. For example, in a database system, as the rate of queries processed increases so does the rate of operations within the operating system, i.e., more I/Os per second (assuming not all data is cached). The analyst can then hypothesize “I expect that the throughput of all components are linearly correlated” – that is, throughput related metrics, i.e., those with units 1/s are correlated. In Listing 3.2, we show how the above hypothesis is specified in Dena. It states that the throughput metrics, i.e., those with units 1/s are expected to be linearly correlated in configurations where the cache_size is less than

```plaintext
Listing 3.1: Invariant Hypothesis

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>HYPOTHESIS HYP-LESS-EQ</td>
</tr>
<tr>
<td>2</td>
<td>RELATION LESS-EQ(x,y) {</td>
</tr>
<tr>
<td>3</td>
<td>&quot;x.name='num_cache_miss' and</td>
</tr>
<tr>
<td>4</td>
<td>y.name='num_cache_get' and</td>
</tr>
<tr>
<td>5</td>
<td>x.component=y.component&quot;</td>
</tr>
<tr>
<td>6</td>
<td>}</td>
</tr>
<tr>
<td>7</td>
<td>CONTEXT {}</td>
</tr>
</tbody>
</table>
```
or equal to 512MB.

The above two examples illustrate the simplicity of the SelfTalk language. We strive to lower the learning curve for analysts to express the behavior of a complex multi-tier system. To achieve this, we provide simple relations (such as the LINEAR and LESS-EQ shown above) along with the system and pre-built hypotheses for common three-tier components, e.g., Apache and MySQL. However, an experienced analyst may define new metrics to monitor, create new relationships to test, and explore new facets of large multi-tier systems. We shall explain the various features of the SelfTalk language in detail in Section 3.3.

3.2.2 The Dena Runtime System

In the following, we provide the steps taken by Dena when the analyst submits a hypothesis to the system.

1. Dena automatically instantiates the hypothesis and generates a (much larger) set of expectations, by enumerating all possible metrics within the metric classes and configurations that match the hypothesis.

2. Dena validates each expectation with experimental data, computes a confidence score per expectation and stores the expectations in a database. The system is now ready for subsequent analysis.

3. The analyst may submit a wide variety of queries to Dena including querying the validity of expectations over components in a sub-part of the system, confidence intervals, number of expectations generated, standard deviations, etc.

Details of Query Execution: Given a hypothesis, Dena creates a list of expectations by iteratively applying the hypothesis for each metric matching the qualifiers, $\vec{Q}$. Next, it selects a function that describes the relationship between the metrics, $R(\vec{Q})$. Then, it evaluates the validity of each expectation using the monitoring data. We describe each step in detail next.

First, Dena creates a list of expectations by applying the hypothesis for each set of metrics matching the qualifiers. For a set of metrics, $\vec{M}$, Dena extracts a subset of metrics $m_i \in \vec{M}$ such that $m_i$ matches all conditions specified in qualifier set $\vec{Q}$. For example, for the query described in Listing 3.2, Dena applies the hypothesis to all
throughput metrics creating a list of expectations. In this list, one expectation would be $\text{EXPECT HYP-LINEAR}(x, y) \ ('name = queries \ per \ sec', 'name = io \ per \ sec')$.

Second, Dena selects a function that matches the relationship described in the hypothesis. We provide a set of pre-defined functions, however, the analyst may also define new relations to use with a hypothesis. For example, if the relationship is $\text{LINEAR}('name = queries \ per \ sec', \ 'name = io \ per \ sec')$ then we match it with a function

$$y_{\alpha, \beta}(x_i) = \alpha x_i + \beta \quad (3.1)$$

and instantiate the expectation.

Third, Dena takes each expectation and fits the function to the monitoring data. The curve is fit using an optimization algorithm, i.e., gradient descent, by varying the free parameters in the function. In particular, for the linear correlation between the database and storage system throughput, the curve fitting algorithm searches for values of $\alpha$ and $\beta$ that minimize the squared error from the measured values. The curve fitting algorithm outputs a confidence score, $\gamma$, between $0 \leq \gamma \leq 1$ representing a goodness of fit where $\gamma = 1$ is a good fit and $\gamma = 0$ is a poor fit. Dena provides the aggregate confidence score for the hypothesis and it allows the analyst to zoom-in to get per-context confidence scores as well. We provide the details on how hypotheses are validated in Section 3.4.

In the following sections, we provide a detailed description of the SelfTalk language and the Dena runtime system.

### 3.3 The SelfTalk Language

In this section, we describe how a hypothesis can be declared in the SelfTalk language and how the generated expectations can be subsequently analyzed using our query language. The SelfTalk language has two types of statements: hypotheses, and queries. The hypothesis states the analyst’s belief about the behavior of the system; it is identified by a unique name, a relation that describes a relationship between metrics, and a context that indicates the configurations affecting the validity of the hypothesis. Dena processes the submitted hypothesis and provides results on whether or not the analyst’s beliefs match the system’s behavior.

To further analyze the results, SelfTalk also allows the analyst to query and check the validity of the expectations; specifically, the analyst can query about the confidence of the expectations (resulting from the expansion of a hypothesis), evaluate the fit under various contexts, and for different sub-components. In addition, the analyst can obtain averages, rank the expectations, and statistically analyze the results computed by Dena. We describe how a hypothesis can be expressed in SelfTalk next; we focus on the different parts: how to specify the metrics,
3.3.1 Hypothesis

The hypothesis expresses the analyst’s belief about the behavior of the system. Each hypothesis is identified by a unique name; this allows the hypothesis to be saved in a database and later retrieved for future querying. The hypothesis describes a relationship (defined as the relation) between metrics (selected from a metric set) for some system configurations (defined as the context). The relation defines a mathematical function describing the relationship between metrics, a set of filters to process the monitoring data (e.g., remove noise), a method to find the best fit, and a mapping to calculate the confidence score from a relation-specific goodness of fit. The relation is identified by a relation name and it may be used in several hypotheses. The relation is evaluated for each combination of metrics contained in a metric set. For example, the analyst may define that she expects the throughput-like metrics to be linearly related; in this case, the relation will be evaluated for each pair of throughput-like metrics from a set of throughput-like metrics. The hypothesis can also specify a validity range—a set of contexts over which the analyst expects the relationship to hold true; the context set is described using a set of metric qualifiers; the context set however also specifies values defining the validity range. In the following, we describe each component of the hypothesis in detail. We leave the details of the processing to Section 3.4.

Metric: The hypothesis describes a relationship between tuples of metrics where each tuple is selected from a metric set (also referred to as the metric class). The metric set, in turn, is constructed by a join of the available metrics (denoted as $M$). In more detail, a hypothesis may define a relationship between two metrics $x$ and $y$, then, the metric set contains tuples of the form $<x_i, y_j>$ chosen from $M^2 = M \times M$ according to the join condition. In general, Dena supports metric sets of more than two metrics. The metric set is constructed from an expression evaluated on each metric’s attributes; the metrics that match the expression are included in the metric set. Each metric is a primitive entity that can be a performance measurement, a configuration setting, or a composite of several base performance metrics. Each metric has several attributes such as its name (e.g., queries_per_sec), the component name (e.g., MySQL) from where it is measured, the location of the component (e.g., hostname of the MySQL instance), and its unit of measurement (e.g., query/sec for throughput). For example, the measure of query throughput, the queries_per_sec metric is defined as

```
METRIC queries_per_sec AS {
    number id,
```
where the MySQL database is running on hostname `cluster101`. Configuration parameters are represented as metrics as well (e.g., `mysql_cache_size`); the configuration metrics are used to establish a context for the hypothesis. In some cases, it is useful to define a composite metric built from a combination of several primitive metrics. The composite metric may be defined persistently within the Dena system or temporarily by inlining the definition with the hypothesis. For example, for the cache, it is useful to define the cache miss-ratio as a composite metric that is computed as the ratio of number of cache accesses (`num_cache_gets`) and the number of cache misses (`num_cache_misses`). The metric set is constructed from the description of metrics given with the hypothesis; Dena selects the metrics by matching the attributes to the conditions specified in the expression (similar to the SQL JOIN and a WHERE clause). The attributes of a metric are optional (except `id` and `name`) and the metric can be thought of a schema-less relation; we use only the specified metric attributes to check a metric for inclusion into the metric set. The expression allows us to specify very broad qualifiers to capture a large set of metrics, or be very specific and capture metrics of a specific component. For example, we can express a relation between a set of throughput metrics, by specifying the qualifiers as "x.unit = ‘1/sec’ and y.unit = ‘1/sec’", or we can express the metrics of a single cache by specifying "x.name = ‘cache_hits’ and y.name = ‘cache_gets’ and x.location = y.location".

**Relation:** The correlation between a set of metrics is described by a relation. The relation includes functions to filter the data, a mathematical function describing the relationship, an error function (e.g., squared error), a method to compute a best-fit (e.g., gradient descent), and a method to compute the confidence score. Many of these functions (e.g., the gradient descent optimizer and the method to compute the confidence score) are independent of the specific relation and may be shared by several relations. We follow an object-oriented paradigm to implement relations; we explain the details of our implementation in Section 3.5.1. To illustrate, we show a SelfTalk code snippet of the linear relation that is provided with the Dena runtime system.

```
DEFINE RELATION linear { 
  PARAMETER a, b : number, 
  INPUT x: number-array, y: number-array, 
  ... 
  FUNCTION confidence 
}```
This code snippet shows the relation containing two parameters and two input data arrays; the parameters refer to the slope and y-intercept of the linear line and the two input arrays correspond to the input and output data values obtained by monitoring the system. We focus on the function to compute the confidence of the relation; the confidence score is a number between 0.0 and 1.0 representing how well the hypothesis fits the monitoring data. In the example, we specify the confidence as the $R^2$ (implemented as a MATLAB script delimited by $) and we also check the residuals before returning the confidence score.

**Context:** The relationship between metrics is influenced by the workload and other system configuration settings – referred to as the *context* of the hypothesis. Therefore, simply fitting the expectations to all measured data would lead to false fits. Consider the expectation EXPECT LINEAR (‘name=queries_per_sec’, ‘name(io_per_sec)’) and assume that we get a 50% hit ratio with a 512MB cache and a 90% hit ratio with a 1GB cache. With different cache sizes, the exact relationship between the metrics (‘queries_per_sec’, ‘io_per_sec’) will be different. In fact, the factor α would be 0.5 for a 512MB cache and 0.9 for a 1GB cache. Specifically, the analyst must provide her belief about the contexts that the hypothesis is sensitive to. A context is simply a list of conditions on a set of performance metrics, workload metrics, or configuration parameters. In Listing 3.2, the context is specified as name=cache_size and value<=512, which states that the analyst expects the hypothesis to hold true only when the cache size is less than 512MB. We also support a wild-card operator, e.g., name=cache_size and value=*, to indicate that cache_size is a configuration parameter that may affect the
fit. In this case, Dena will evaluate the expectation for each setting of the configuration separately.

### 3.3.2 Query

Dena expands the hypothesis submitted by the analyst into expectations, fits each expectation to the monitoring data, and stores the results in a database; these results can be further analyzed by submitting queries written in SelfTalk. The analyst can query about the confidence of the expectations that result from the expansion of the hypotheses, evaluate the fit under various contexts and for different sub-components. We categorize the queries into two types: i) queries that focus the analysis on particular components, configurations, or confidence values, and ii) queries that modify the presentation of the results by ordering them based on confidence score, grouping them by particular metrics, or grouping them by the configuration type.

The general syntax of a SelfTalk query is

```
1. QUERY <HYP-NAME>
2.   [METRIC <METRICS-SET>]
3.   [CONTEXT {<CTX-SET>}]  
4.   [CONFIDENCE {<|>|>=|=|<= <VALUE>>}]
5.       |{<IN> <RANGE>}]
6.   [ORDER BY CONFIDENCE [ASC|DSC]]
7.   [RANK BY CONFIDENCE [ASC|DSC]]
8.   [GROUP BY METRIC <METRIC>. . .<METRIC>]
9.   [GROUP BY CONTEXT <CONTEXT>. . .<CONTEXT>]
```

A query consists of three parts: i) the preamble – we need to specify the name of the hypothesis being queried, e.g., the hypothesis name (shown in line 1), ii) the query focus – we narrow the analysis by specifying conditions on the metric set, the context set, and the confidence score (lines 2-5) and iii) the presentation of results – the results may be displayed by controlling the ordering based on the confidence score, and by grouping using a certain metric attributes or contexts (lines 6-9). We present the details of how queries enable analysis of the results using two examples next.

All queries include a hypothesis name; the hypothesis name is used to find the results stored by Dena in the database. If no options are specified, the results of all expectations that are generated from the hypothesis are returned — that is, the results all possible expansions (expectations) of the metric set and context set declared in the hypothesis (this is equivalent to the SELECT * construct in SQL). SelfTalk allows the fine-grained analysis to be done with ease by restricting the analysis to certain sub-components and for certain contexts. For example, the analyst may issue
CHAPTER 3. ANOMALY DETECTION BASED ON INVARIANT VALIDATION

QUERY HYP-LINEAR
METRIC (x, y) {
  "x.component='MySQL' and x.unit='1/sec'
  and
  y.component='Akash' and y.unit='1/sec'"
}

CONTEXT (a) {
  "a.name='mysql_cache_size' and a.value=512"
}

CONFIDENCE > 0.9

that returns results from expectations of the linear hypothesis (named HYP-LINEAR) for throughput-like metrics measured at the Akash storage server and MySQL only for configurations where the size of the MySQL cache is configured to 512MB and those expectations with a confidence score greater than 0.9.

In addition to allowing focused analysis of the results, SelfTalk allows the analyst to control the presentation of the results of a query by grouping, ordering, and ranking. We can analyze the effect of changing the size of the MySQL cache on the throughput by stating

QUERY HYP-LINEAR
METRIC (x, y) {
  "x.component='MySQL' and x.unit='1/sec'
  and
  y.component='Akash' and y.unit='1/sec'"
}

ORDER BY CONFIDENCE DSC
GROUP BY CONTEXT (a) {
  "a.name='mysql_cache_size' and a.value=*"
}

to return expectations from the execution of the linear hypothesis (named HYP-LINEAR) for throughput-like metrics collected from MySQL and the Akash storage server, grouped by MySQL cache configurations (where the confidence scores are computed as the average for each cache configuration) and sorted by the confidence score in descending order.
3.4 Validating Expectations

Dena expands the hypothesis posed by the analyst to generate a larger set of expectations by enumerating all possible metrics and configurations that match the hypothesis. In this section, we describe the steps taken by Dena to validate each expectation with the monitoring data and compute the confidence score.

3.4.1 Overview

An expectation is validated by evaluating how well the relationship described by the hypothesis applies to the monitoring data. At its core, we apply statistical regression techniques to fit a function (describing the relationship between metrics) and evaluate the goodness of fit. While statistical regression techniques have been studied in great detail elsewhere [89], the three main challenges exist in the implementation of a generic engine; we need to (1) process monitoring data collected from many different sources, (2) evaluate various relationships on the monitoring data, and (3) compute a mapping from the relationship specific goodness of fit to a human-understandable confidence score.

The first challenge arises from the fact that monitoring data from a component contains noise and that monitored values from multiple components may not be aligned in time. Thus, we first filter the data to make it suitable for statistical regression; filtering removes the outliers in the collected data and aligns the time-series data. After filtering, we can evaluate if the relation matches the monitoring data. The second challenge is that the statistical regression techniques differ for different types of relations; while at the heart of all expectations is a mathematical function describing a relationship between a set of monitored metrics, the method of fitting the function differs from closed-form solutions (e.g., for linear regression) to iterative methods such as gradient descent. Finally, we need to compute a confidence score – a human understandable output between 0.0 (low confidence) and 1.0 (high confidence) from the relation-specific goodness metric. To aid in the design of a generic engine, we evaluate a set of commonly asked questions by analysts and build a taxonomy of relations. In the following, we describe the taxonomy of relations and describe each of the steps in more detail. Then, we provide a list of sample relations used to evaluate the behavior of a multi-tier system.

3.4.2 Taxonomy

A relation describes a mapping between several metrics. Each relation specifies a function \( \hat{y} = \hat{f}(x) \) that describes how two metrics \( x \) and \( y \) are expected to be correlated. The relationship may be comparisons – where the mapping between \( x \) and \( y \) is a boolean operator e.g., \( y < x \) or regressions – where the mapping between \( x \) and \( y \) is a mathematical function e.g., \( y = ax + b \). In addition, each of the relationships may be time dependent e.g., \( \hat{y}_t = \hat{f}(x_t) \) or time-independent.
We classify the relations into different categories using the above criteria as shown in Figure 3.1. The relations are first classified into two categories: regressions and comparisons. The relations classified into regressions are functions that describe a mathematical relationship between several metrics. An example of a regression relationship is a linear relationship between two metrics; the function mapping $x$ to $y$ is described by $\hat{y}(x) = \alpha x + \beta$. The validity of these relations can be evaluated using statistical regression techniques. The second relation type is a comparison where the mapping between two metrics is a comparison operator ($<$, $>$, $=$, $\leq$, $\geq$). In this case, directly applying statistical regression techniques is difficult. Thus, we evaluate the validity of these relations using simple counting; we validate the relation by counting the fraction of points where the comparison holds true. Each of the above two categories (regressions and comparisons) can be applied to time-dependent or time-independent data. Time-dependent relations treat the input as time-series in which the relation between input metrics are considered through time; the input data to the relation is tuples of metric values with same time-stamps. On the other hand, time-independent relations treat the data as an unordered list. We explain the details next.

### 3.4.3 Evaluating Expectations

The evaluation of an expectation consists of three steps: (1) collect and filter monitoring data, (2) apply statistical regression and evaluate for monitoring data, and (3) compute the confidence score.

**Step 1 – Filtering Monitoring Data:** The monitoring data collected from components have two sources of error: (1) noise in the data collected from one component, and (2) mis-alignment of data collected from multiple components. We filter the data values before evaluating the relationship.

The noise in the monitoring data is seen as outliers in the data. The outliers occur when data is collected from components during their initialization phases either at start-up or after a configuration change, and due to
interference from background tasks. One such example is the measurement of the cache_hits (the number of cache hits) and cache_misses (the number of cache misses) from a cache. During the initialization phase (i.e., cache warm-up), the cache misses are high as many of cache accesses experience cold misses since the cache is empty. However, as the cache warms up, the number of cache misses decreases steadily (conversely the number of cache hits increases steadily) until the values reach steady state. Similarly, infrequent background tasks from the operating system or transient network bottlenecks introduce noise in the measurements as well. We filter these outliers before applying statistical regression. The analyst can instruct Dena to apply any filtering technique. We choose to use percentile filtering due to its simplicity. Percentile filters are generic; they make no assumption about the distribution of data other than that the number of samples is large enough to cover most regions of the underlying distribution. We use percentile filtering to trim the top $t\%$ and the bottom $b\%$ of sampled data. By removing these samples, the percentile filter keeps the samples which form the majority in the distribution. Based on experience and insight about the process of collecting monitoring data, the analyst may specify filtering thresholds $t\%$ and $b\%$ thereby overriding the default values. The filtering process is different for time-independent relations; in these, we perform percentile filtering per configuration value rather than on the entire dataset.

Time-series data pose an additional challenge because the data measured at different components may be misaligned due to clock skew as well as due to causality between components. We evaluate time-series data by matching (i.e., joining in the database terminology) the sampled values using the timestamp. Causality between components can also account for some misalignment between the sampled metrics. For example, a change in the workload is reflected at the metrics collected at the higher layers (e.g., the database) before it is seen in the metrics collected in the lower layers (e.g., disk). While there are various sophisticated methods for aligning time-series data, we find that simple techniques of grouping values using a coarser-time granularity and using moving average filters work well; for example, we align the data values by grouping them into a coarse timestamp granularity (e.g., 10 seconds). We also use a moving-average filter. A moving average is used to analyze a set of data points by creating a series of averages of adjacent subsets of the full data set; this smooths out short-term fluctuations while maintaining the long-term trends. Aligning time-series data by estimating the clock skew and delay between components is an area for improvement; we leave this optimization as future work.

**Step 2 – Performing Regression:** After filtering the monitoring data, we perform statistical regression to evaluate how well the hypothesis fits the measured values. We find the best values for free parameters to reduce the squared error between the hypothesis and the measured values. For example, consider the linear relation

$$
\hat{y}_{a,\beta}(x_t) = \alpha x_t + \beta
$$

with two free parameters $\alpha$ – the slope of the line, and $\beta$ – the y-intercept of the line. The best fit of the relation
to the measured data is obtained when the squared-error between the predicted values and the measured values is
minimized. We define the error (i.e., how the relation deviates from the measured values) as
\[ \xi(\alpha, \beta) = \sum_{x,y} (y - \hat{y}_{\alpha,\beta}(x))^2 \] (3.3)
and we find the best-fit of the relation by minimizing the squared error by mapping the problem of reducing
the squared error as an optimization problem and use standard optimization techniques such as gradient descent
(using the partial derivatives if given) to find the best parameter values. In some cases, the best parameter values
are obtained from closed-form solutions (such as for linear regression); we opt for the closed-form solution
rather than iterative search in these cases.

**Step 3 – Computing the Confidence Score:** After applying statistical regression and optimizing the free
parameters, we evaluate how well the relation describes the data and report the *confidence score*. The *confidence
score* is a human-understandable number between 0.0 and 1.0, which indicates a poor and good fit respectively.
The evaluation of the confidence score is dependent on the relation – whether the relation is a *comparison* or
*regression*.

For the *comparison* relations, the confidence score is the fraction of data when it holds `true`; we count the
number of times the comparison evaluates to `true` and divide by the total number of monitoring data points. For
*regression* functions (i.e., those with a mathematical relationship), we use the coefficient of determination, \( R^2 \), to
compute the confidence score. The \( R^2 \) is a fraction between 0.0 and 1.0. An \( R^2 \) value of 0.0 indicates that the
function does not explain the relationship between the two metrics. Assuming a relation is defined as \( \hat{y} = \hat{f}(x) \),
the coefficient of determination is defined as
\[ R^2 = 1 - \frac{SS_{err}}{SS_{tot}} \] (3.4)
\[ SS_{err} = \sum_i (y_i - \hat{f}(x_i))^2 \] (3.5)
\[ SS_{tot} = \sum_i (y_i - \bar{y})^2 \] (3.6)
where \( SS_{err} \) is the residual sum of the squares, \( SS_{tot} \) is the total sum of squares, and \( \bar{y} = mean(y) \). However,
simply using \( R^2 \) to evaluate the fit may be incorrect. To better evaluate the fit, we perform a secondary test using
the *residuals* of the regression; the residuals are the vertical distances from each point in the fitted line to the
monitoring data. A good fit has the residuals equally above and below the fitted line. If the residuals are not
randomly scattered – indicating a systematic deviation from the fitted line then, the \( R^2 \) value may be misleading;
thus we report that the fit has a low confidence score.
3.4.4 Validating Performance of a Multi-tier Storage System

In this section, we provide a sample of hypotheses that we issue to understand and validate the behavior of a multi-tier storage system consisting of a MySQL database using a virtual volume hosted on a storage server. The details of the storage system are given in Section 3.5.2. We choose one or two hypotheses from each of the categories we describe in the relation taxonomy. For each hypothesis, we provide the high-level question the analyst is probing, the underlying regression/comparison function tested in the hypothesis, the filtering applied to the monitoring data, and the optimization algorithm used to find the best fit.

**Time-dependent Regression – Linear/Little:** The **LINEAR** hypothesis is one of the simplest hypothesis that an analyst can issue to Dena; we issue this hypothesis to diagnose traffic patterns along the storage path. Specifically, as an analyst, we ask the question – “I expect the throughput measured at the storage system to be linearly correlated with the throughput measured at MySQL” or more generally “I expect the throughput metrics along the storage path to be linearly related” with the belief that as we increase the load at the MySQL database, the load on the underlying storage server will increase correspondingly. The linear relation is defined as

\[
\hat{y}_{\alpha, \beta}(x_t) = \alpha x_t + \beta
\]

with two free parameters: \(\alpha\) and \(\beta\). We filter the time-series data by first removing the outliers using percentile filtering and then smoothing the values with a moving average filter. The line is fit to the monitoring data using linear regression and we use the coefficient of determination \((R^2)\) as the confidence score. We further verify the fit using the residuals to determine if the data does not systematically deviate from the hypothesis. If the residuals are not valid, we report that the hypothesis is not a good fit.

**Dena** can incorporate results from models, such as those derived from operational laws, to verify the behavior of a multi-tier system; an example of this is the **LITTLE** hypothesis that defines a relationship between throughput and latency using Little’s law [106]. Little’s law states that if the system is stable then, the response time and throughput are inversely related; we issue this hypothesis to verify that the behavior of the system adheres to the behavior explained by operational laws; a stable system follows these laws. For example, the analyst can express her belief in operational laws by making a high-level hypothesis that “I expect the throughput measured at the storage system is inversely correlated with the latency measured at MySQL”. For an interactive system, such multi-tier storage systems, Little’s law is expressed as

\[
\hat{X}_{N, Z}(R_t) = \frac{N}{R_t + \mathcal{Z}}
\]

with two free parameters: \(N\) and \(\mathcal{Z}\), which are number of clients and average think time respectively, and \(X_t\) and \(R_t\) denoting throughput and response time. Similar to the processing of **LINEAR** relation, we filter the data.
by first removing the outliers using percentile filtering and then smoothing the values with a moving average filter. The curve is fit to the monitoring data using gradient descent optimization, and we use the coefficient of determination ($R^2$) as the confidence score. We further verify the fit using the residuals to determine if the data does not systematically deviate from the hypothesis; if the residuals are not valid, we report that the hypothesis is not a good fit.

Time-independent Regression – Quanta: Our storage system uses the quanta-based scheduler to divide the storage bandwidth among several virtual volumes. The quanta-based scheduler partitions the bandwidth by allocating a time quantum where one of the workloads obtains exclusive access to the underlying disk. For modeling the quanta latency, we observe that the typical server system is an interactive, closed-loop system. This means that, even if incoming load may vary over time, at any given point in time, the rate of serviced requests is roughly equal to the incoming request rate. Then, according to the interactive response time law [106]:

$$L_d = \frac{N}{X} - Z$$  \hspace{1cm} (3.9)

where $L_d$ is the response time of the storage server, including both I/O request scheduling and the disk access latency, $N$ is the number of application threads, $X$ is the throughput, and $Z$ is the think time of each application thread issuing requests to the disk. We then use this formula to derive the average disk access latency for each application, when given a certain fraction of the disk bandwidth. We assume that think time per thread is negligible compared to request processing time, i.e., we assume that I/O requests are arriving relatively frequently, and disk access time is significant. Then, through a simple derivation, we arrive at the following formula

$$L_d(\rho_d) = \frac{L_d(1)}{\rho_d}$$  \hspace{1cm} (3.10)

where $L_d(1)$ is the baseline disk latency for an application, when the entire disk bandwidth is allocated to that application. This formula is intuitive. For example, if the entire disk was given to the application, i.e., $\rho_d = 1$, then the storage access latency is equal to the underlying disk access latency. On the other hand, if the application is given a small fraction of the disk bandwidth, i.e., $\rho_d \approx 0$, then the storage access latency is very high (approaches $\infty$). The QUANTA hypothesis expresses the above belief from the operational law model where we expect the storage access latency of the application to be inversely related to the allocation time fraction. The QUANTA hypothesis uses the inverse relationship that is described as

$$\hat{y}_{a,\beta}(x) = \frac{\alpha}{x^\beta}$$  \hspace{1cm} (3.11)
where the waiting time at the scheduler \( (y) \) is inversely related with the time fraction \( (x) \) given to the application. We filter the latency values using the percentile filter and average the samples (for each quanta setting) before performing regression. We find the best-fit for the free parameters using gradient descent and we use \( R^2 \) as the confidence score and use the residuals as a secondary check.

**Time-dependent Comparison – Less/EQ:** The LESS/EQ hypothesis is used to answer many storage questions. For example, the analyst can check on a configuration parameter — “I expect the current size of the cache is less than or equal to the maximum size (as defined in the configuration)” or check on a performance metric — “I expect the latency (e.g., response time) measured at higher level components (MySQL) is higher than the latency measured at the lower level components (disk)” . We remove the outliers using percentile filtering and use a moving average filter to synchronize the samples over time. There is no regression step and we report the confidence score as the fraction of samples where the comparison \( (\leq) \) holds true.

**Time-independent Comparison – MRC/Constant:** The miss-ratio curve \( (\text{MRC}) \) relation describes the behavior of a cache; it states that the cache miss-ratio (i.e., the ratio of cache misses to the cache accesses) is a monotonically decreasing curve with respect to the cache size. We capture this relationship in two ways: by comparing to a user-provided miss-ratio function or systematically checking that the curve is indeed monotonically decreasing. In the first case, the analyst may provide the expected miss-ratio curve from a model (i.e., using Mattson’s stack algorithm \cite{77}) or from a cache simulator; with either approach, we are given a list of tuples of the form \( \langle c, m \rangle \) (where \( c \) is the cache size and \( m \) is the miss-ratio) and we evaluate the confidence using \( R^2 \). In the second method, for each cache size \( c \), we obtain the values of the miss-ratio and apply the percentile filter; the filtering concentrates the miss-ratio samples into a cluster (for each cache size \( c \)). Then, we average the miss-ratios and use the resulting list \( \langle c, \bar{m} \rangle \) of tuples (where \( \bar{m} \) is the average of the miss-ratios for cache size \( c \)) to sort by cache sizes in ascending order and verify that the miss-ratio keeps decreasing (or remains flat) as the cache size is increased. We count the fraction of times the comparison holds true and report it as the confidence score.

The versatile \text{CONST} hypothesis checks if the values of a metric are constant; we use this relation to issue hypothesis of the form “\text{I expect that the size of the cache (i.e., the number of items stored in the cache) remains constant}”. We note that there is a small fraction of time (during start-up) when the size is not equal to the capacity which is filtered by the percentile filter. We filter the data using percentile filter to remove outliers and return high confidence if samples are almost constant – that is, the variation in the values is within a small ratio of its mean; we compute the ratio of the mean of \( x \) and divide by the standard deviation. If the ratio is less than a threshold, we report a confidence score of 1, else we report a confidence score of 0.
3.5 Testbed

In this section, we describe the implementation of *Dena* and our experimental multi-tier infrastructure consisting of a MySQL database running on a virtual storage system called *Akash*.

3.5.1 Prototype Implementation

The *Dena* runtime system is composed of multiple parts: a front-end consisting of the *SelfTalk* parser, a core regression engine, and a database backend storing the monitoring data. The monitoring data is collected from existing software; we use built-in instrumentation such as the MySQL/InnoDB monitor to get statistics from the database, *vmstat* and *iostat* to obtain statistics from the operating system, and built-in instrumentation from our storage server. Capturing the statistics has no runtime overhead on the system because they are part of operational metrics that are exposed by the system by default. We implement the core of the statistical regression algorithms using MATLAB utilizing JDBC to fetch data from the backend database. We provide simple relations that can be utilized by an analyst new to *Dena*; this includes all the relations we describe in Section 3.4.4 plus we provide relations describing exponential and polynomial curves and all boolean comparisons.

The analyst can specify the hypothesis at the command-line or by referring *Dena* to a file; given a hypothesis *Dena* parses the details and expands the hypothesis to all possible expectations. *Dena* instantiates a new object for each expectation, obtains the data from the database, fits the relation to the monitoring data, and computes the confidence score. When the fitting is complete, the details of the hypothesis, the set of expectations, the final fitted values of the free parameters, and the descriptions of the contexts are stored into the database for future analysis.

3.5.2 Platform and Methodology

Our evaluation infrastructure consists of two machines: (1) a database server running OLTP workloads and (2) a storage server running *Akash* [97] to provide virtual disks. *Akash* is a virtual storage system prototype designed to run on commodity hardware. It uses the Network Block Device (NBD) driver packaged with Linux to read and write logical blocks from the virtual storage system, as shown in Figure 3.2. The storage server is built using different modules:

- **Disk module**: The disk module sits at the lowest level of the module hierarchy. It provides the interface with the underlying physical disk by translating application I/O requests into *pread()*/*pwrite()* system calls, reading/writing the underlying physical data.

- **Quanta module**: The quanta module partitions the disk bandwidth using a quanta-based I/O scheduler [97]. The scheduler provides a fraction of the disk time to each workload sharing the disk volume.
Figure 3.2: **Testbed**: We show our experimental platform. It consists of a storage server (*Akash*) and a storage client (DBMS) connected using NBD.

- **Cache module**: The cache module allows data to be cached in memory for faster access times.

- **NBD module**: The NBD module processes I/O requests, sent by the client’s NBD kernel driver, to convert the NBD packets into calls to other *Akash* server modules.

We use three workloads: a simple micro-benchmark, called UNIFORM, and two industry-standard benchmarks, TPC-W and TPC-C. We run our Web based applications (TPC-W) on a dynamic content infrastructure consisting of the Apache web server, the PHP application server and the MySQL/InnoDB (version 5.0.24) database engine. We run the Apache Web server and MySQL on Dell PowerEdge SC1450 with dual Intel Xeon processors running at 3.0 Ghz with 2GB of memory. MySQL connects to the raw device hosted by the *Akash* server. We run the *Akash* storage server on a Dell PowerEdge PE1950 with 8 Intel Xeon processors running at 2.8 Ghz with 3GB of memory. To maximize I/O bandwidth, we use RAID-0 on 15 10K RPM 250GB hard disks. Non-web applications (TPC-C) utilize the same MySQL and storage server instances; however, they do not use the machine running the Apache web server. The monitoring data is collected from the underlying operating system (using Linux utilities `vmstat` and `iostat`), the MySQL database, and the *Akash* storage server. The collected metrics are timestamped using `gettimeofday()`.

Specifically, we use the metrics that were collected over a period of 6 months [97]. The collected data includes storage-level metrics (from the *Akash* storage server), database-level metrics (gathered by instrumenting MySQL), and OS-level metrics (using `vmstat`). We collected the data for two physical machines, i.e., the database machine and the storage machine, and for four applications, i.e., four virtual disk volumes and database instances. The collected metrics, after pruning, result in over 10GB of data represented as flat files; we load these files into the database for analysis.
3.6 Results

We evaluate the efficacy of Dena to validate overall system behavior and to understand per-component behavior. To achieve this, we issue broad high-level hypotheses describing the relationships in a multi-tier storage system and check the validity of these relationships. Next, we issue specific queries to provide insights into the behavior of a specific component and also one component's effect on other components within the multi-tier system. Then, we present additional results studying cases where there is a mismatch between the analyst’s belief and the monitoring data. Finally, we present measurements calculating the cost and time breakdown of executing a hypothesis.

3.6.1 Understanding the Behavior of the Overall System

We issue several broad high-level hypotheses to check the overall behavior of the system. We present the correlations that Dena discovers for three simple hypotheses: (1) LINEAR – expects that metrics of the same type are linearly correlated, (2) LESS/EQ – states that round-trip latency is additive across layers and (3) LITTLE – states that throughput and latency adhere to the Little’s law. Table 3.1 shows the number of expectations generated for each hypothesis for all contexts. Dena generates the expectations automatically for a given hypothesis. Figure 3.3 shows the correlations discovered by Dena in a graph where the nodes represent metrics and the edges indicate a correlation. To simplify the presentation, we only show metrics related to the throughput and latency for each module. In addition, we only show results where we configure the cache to 1 GB resulting in a 50% miss-ratio and allocate the entire disk bandwidth to the application. We explain the correlations discovered for the LESS/EQ and LITTLE in detail next.

Table 3.1: Expectations. We show the number of expectations generated for each high-level hypothesis.

<table>
<thead>
<tr>
<th>Hypothesis</th>
<th>Expectations</th>
<th>Avg. Confidence</th>
</tr>
</thead>
<tbody>
<tr>
<td>LINEAR</td>
<td>3072</td>
<td>86%</td>
</tr>
<tr>
<td>LESS/EQ</td>
<td>3488</td>
<td>98%</td>
</tr>
<tr>
<td>LITTLE</td>
<td>3290</td>
<td>92%</td>
</tr>
</tbody>
</table>

For the LINEAR hypothesis, shown in Figure 3.3(a), we find two clusters of metrics: a set of throughput related metrics and a set of latency related metrics. First, we see that the set of throughput metrics are linearly correlated. This is expected as the storage is configured as a single path from the NBD module to the disk module (see Figure 3.2). The cache and quanta modules do not affect the linear correlation between the throughput seen in the NBD module (nbd_enter) and the disk module (disk_enter) because while the cache causes less I/Os to be issued to disk, an increase in the rate of I/O requests entering the storage system still results in a corresponding increase in the rate of disk I/Os. Similarly, latency across components is linearly correlated as well except the quanta module; it controls the number of requests entering disk leading to an additional queuing delay between the disk_latency and the quanta_latency breaking the linear relationship across latencies [97].
Figure 3.3: Correlations. We show the pairwise correlations we discover for different analyst hypotheses in the above graph. The nodes represent different metrics and the edges show the correlation. The above results were gathered with a 1GB cache resulting in a miss-ratio of 50%, and the entire disk bandwidth was allocated to the application.

We develop the LESS/EQ hypothesis by using the information of the structure of Akash which allows us to hypothesize that latencies (similarly throughput) measured in some modules are less than the latencies measured in other modules. Figure 3.3(b) shows our results using a directed graph where the arrowhead points from the smaller metric to the larger metric. For example, the cache module sits above the quanta module and forwards requests only on cache misses. Therefore, with a 50% miss-ratio, the latency at the cache module is less than...
the quanta module. This is shown as an arrow from \textit{cache\_latency} to \textit{quanta\_latency}. Conversely, the number of requests entering the quanta module is less than the number of requests entering the cache module, shown as an arrow from \textit{quanta\_enter} to \textit{cache\_enter}.

As Akash is a closed-loop storage system, we hypothesize that performance adheres to Little’s law \cite{106} — that is, the throughput and latency metrics follow the interactive response time law and thus are inversely proportional. Figure 3.3(c) shows that indeed the system complies with Little’s law as the throughput and latency metrics are indeed correlated. \textit{disk\_latency} is not correlated with Little’s law as the quanta module self-adjusts its scheduling policy to varying disk service times \cite{97} leading to a weak correlation with the \textit{disk\_latency}.

### 3.6.2 Understanding Per-Component Behavior

Next, we explore the behavior of different storage server components by studying the correlations found using different hypotheses. We focus on the two major components: the cache and the quanta scheduler modules within Akash. Then, we present results showing how Dena can be used to study interactions between multiple components as well; to illustrate this, we focus on the effect of cache inclusiveness in multi-tier caches.

**Understanding the Cache:** We study the effect of caching on the performance of the storage system by issuing several hypotheses that provide an insight into its behavior: \textit{MRC} – indicates the analyst’s belief that the cache performance will improve (i.e., its miss-ratio will decrease) as the size of cache is increased, \textit{LESS/EQ} – states that caching improves performance by reducing latency where the latency to access items from the cache is lower than the latency of accessing items from the underlying disk, and \textit{LINEAR} – states the belief that since the cache size impacts performance, the linear relation between metrics must account for the size of the cache as a context. We evaluate these beliefs using the UNIFORM workload which has a miss-ratio of 75\% with a small cache (256MB), 50\% with a medium cache (512MB) and a 12\% with a large cache (896MB). Figure 3.4 shows the results of the \textit{MRC} and \textit{LINEAR} hypotheses. The results from the TPC-W workload are similar.

Figure 3.4(a) shows the miss-ratio for the UNIFORM workload. As expected, the miss-ratio is monotonically decreasing – a straight line from approximately 1.0 (many misses) with a small cache to near 0.0 (many hits). Dena computes a confidence score of 0.99 for the miss-ratio curve. Regardless of the cache size, caching provides a benefit in terms of performance. This improvement can be checked using the \textit{LESS/EQ} hypothesis; Dena reports a confidence score of 1.0 for all cache sizes indicating that the throughput measured at the cache is higher than the throughput at the underlying disk and the latency at the cache is lower than the latency of fetching data from disk.

The detailed impact on the performance from different cache sizes can be obtained by issuing the \textit{LINEAR} hypothesis as seen in Figures 3.4(b)-3.4(c). Each plot shows three lines corresponding to three cache sizes: a small cache (shown in red with squares), a medium cache (shown in green with triangles), and a large cache
(shown in blue with circles). The points are the samples (before percentile filtering) obtained through monitoring and the line is the best-fit of the relation described in the hypothesis. The plots show that performance can indeed be improved by increasing the size of the cache; the throughput ratio between the cache and the disk (i.e., the factor of improvement) is 1.25, 2, and 8 for small, medium, and large cache sizes respectively. Similar factors are seen in the reduction of the access latency at the cache and the underlying disk latency.

**Understanding the Quanta Scheduler:** The quanta scheduler is the mechanism *Akash* uses to proportionally allocate the disk bandwidth among multiple storage clients. As we describe in Section [3.4.4](#), the effect on performance can be modeled using operational laws. In this case, we observe that the *Akash* is a closed-loop system where the rate of serviced requests is roughly equal to the incoming request rate. Then, by using the *interactive response-time law*, we derive the relationship that the latency as seen from the quanta module varies inversely with fraction of the disk bandwidth allocated to the workload – that is, as the fraction of disk bandwidth is halved, the per-request latency doubles.

Figure [3.5](#) presents the results obtained from *Dena* for the UNIFORM workload. It shows three curves showing the results for the small, medium, and large cache sizes. In addition, we plot the measured values of the quanta latency for comparison. The results show that our belief that the latency varies inversely to the disk bandwidth fraction is correct; the fitted curve closely matches the observed values resulting in confidence scores of 0.94, 0.94, and 0.93 for the small/medium/large caches respectively. Using the QUANTA hypothesis allows us to understand the disk performance as well. Specifically, *Dena* shows that the confidence score for the large cache is slightly smaller than the small and medium cache sizes. The reason is that there is a higher variability of the average disk latency when (i) the underlying disk bandwidth isolation is less effective due to frequent switching between workloads and (ii) disk scheduling optimizations are less effective and reliable due to fewer requests in the scheduler queue. However, even with this variability, the underlying relationship is still inverse leading *Dena* to report a high confidence score.

**Understanding Two-tiers of Caches:** In a multi-level cache hierarchy using the standard (uncoordinated) LRU replacement policy at all levels, any cache miss from cache level $q_i$ will result in bringing the needed block into all lower levels of the cache hierarchy, before providing the requested block to cache $i$. It follows that the block is redundantly cached at all cache levels, which is called the *inclusiveness* property [108]. Therefore, if an application is given a certain cache quota $p_i$ at a level of cache $i$, any cache quotas $p_j$ given at any lower level of cache $j$, with $p_j < p_i$ will be mostly wasteful. We can verify this behavior using two hypothesis based on the MRC hypothesis. Due to cache inclusiveness, the analyst expects that by increasing the size of the first-level cache (i.e., the MySQL buffer pool) the performance of the second-level cache (i.e., the miss-ratio at the storage server cache) steadily decreases due to lower temporal locality.

We perform the analysis by stating that the relationship between the miss-ratio at the storage cache and the size of the MySQL buffer pool size is monotonically increasing; the context of the hypothesis is the storage cache
Figure 3.4: **Understanding the Cache Behavior:** We look at the impact of caching on the performance of the storage server by studying the miss-ratio curve and comparing the throughput and latency across the cache module within *Akash.*
Figure 3.5: Understanding the Quanta Behavior: We see that the impact of the quanta scheduler is inverse where halving the disk bandwidth fraction leads to a doubling of the quanta latency.

size. Given this hypothesis, Dena presents these results grouped by each storage cache size. We present the results graphically for the TPC-W workload; the results from TPC-C are similar. Figure 3.6 shows this behavior for three different storage cache sizes: small (128MB), medium (512MB), and large (896MB) where the lines indicate the best-fit regression and the points are measured values. For the small storage cache (shown in blue with squares), we see that the miss-ratio is high at 80% for small MySQL buffer pool sizes but quickly increases to 100% for medium to large MySQL buffer pool sizes. For a large storage cache (shown in red with circles), the effect is more clear; the miss-ratio for a small MySQL cache is less than 25% but the miss-ratio worsens steadily as the MySQL cache is increased where it crosses 50% after 512MB of MySQL buffer pool and over 90% for very large MySQL cache sizes.

3.6.3 Understanding Mismatches between Analyst Expectations and the System

There can be a mismatch between the analyst’s beliefs and the monitoring data; this can occur either due to a fault in the system or from a misunderstanding of the system by the analyst. In either case, Dena reports low confidence scores and the analyst may probe deeper by issuing different hypotheses to diagnose faults or to improve her understanding of the system. In the following, we present three cases of mismatch; we test for cases where (i) the system is faulty – we induce a fault in the cache resulting in errors in the cache replacement policy, (ii) the hypothesis is faulty – we hypothesize that the behavior of the quanta scheduler is linear, and (iii) the context is faulty – we hypothesize the metrics of the same type are linearly correlated but fail to provide the context information that the size of the storage cache may affect the relationship.
System is Faulty: In the first case, we show results showing how Dena can be used to detect a fault in the system; we detect a fault in the cache replacement policy using the MRC hypothesis which states that “I expect the cache misses to decrease monotonically with increasing cache size”. We run the UNIFORM workload for this experiment; in an earlier case, we have shown that the UNIFORM workload has a straight line as the miss-ratio curve, shown in Figure 3.4(a) and that with a fault-free cache replacement algorithm, the curve is indeed monotonically decreasing. Now we induce a fault in the cache replacement algorithm that reduces caching benefit; it has more cache misses than expected for some cache sizes as shown in Figure 3.7(a). Due to the fault, Dena is not able to validate the relationship using the monitoring data; this leads Dena to report a very low confidence score of 0.24. This scenario highlights one use-case where the analyst is confident in her hypothesis and thus can conclude that the system is faulty.

Hypothesis is Faulty: Another case where there is a mismatch between the analyst and the system is if the analyst’s belief is incorrect; we test a case by issuing the hypothesis that we expect the “latency of the quanta module is linearly related with the disk bandwidth fraction”. During the design phase of Akash, we made a similar assumption; we noticed that the throughput of the storage system varies linearly with the disk bandwidth fraction (by applying Little’s law) and incorrectly concluded that the effect on latency is linear as well. We have shown that the relationship is indeed inverse earlier in Figure 3.5 the error is noticed by Dena, as shown in Figure 3.7(b) where the expected line does not match the monitoring data. In this case, Dena reports a confidence score of 0.8. This scenario describes the second use-case where the analyst initiates a dialogue to understand the behavior of the system by issuing hypotheses (correctly or incorrectly) and obtaining feedback on its validity.
Figure 3.7: Different Errors: Dena does not expect the analyst to issue correct hypotheses or the system to behave correctly. In both cases, there is a mismatch between the analyst and the system leading to low confidence scores. We show three such cases.
Context is Faulty: In the last case, we re-issue the LINEAR hypothesis but fail to identify that the size of the cache may affect the validity of the hypothesis. With an incorrect context, the relation cannot be fit; as Figure 3.7(c) shows, the data values form several lines with different slopes and y-intercepts and no single line satisfies the monitoring data. With an incorrect context, the best-fit of a line is a null fit and the confidence score is 0.0.

3.6.4 Cost of Hypothesis Execution

We also evaluate the cost of executing a hypothesis by measuring the time taken to fetch the data from the database and the time needed to perform statistical regression.

Our knowledge base is stored in a relational DBMS (PostgreSQL) and we use JDBC to fetch the data to be used by MATLAB for data processing. Our analysis shows that a majority of time is spent fetching the data from the DBMS and not in data processing (MATLAB). However, we also performed further analysis by considering monitoring data over longer time intervals (6 months) thereby stressing MATLAB. In this longer time interval, with roughly 1.5M samples, the time spent inside MATLAB is under 3 seconds. Similarly, in this case, the majority of the time is spent fetching the data from the DBMS/Disk.

In more detail, Figure 3.8 presents our results for queries accessing up to a week of monitoring data. It shows that a large fraction of the time is spent fetching the data from the database and a small fraction spent doing statistical analysis. Specifically, our results show that it takes roughly 1 to 1.5 seconds (average) to fetch the data for an expectation and less than 40ms to find the best-fit. The computation cost is the least for comparison relations; these perform simple counting thus require less than 5ms to report the confidence score. The regression cost is higher as we need to fit the line to the monitoring data; the time needed to find the closed-form solution for LINEAR is 25ms and the time needed for QUANTA (inverse) is 39ms on average.

3.7 Summary

We introduce SelfTalk – a declarative high-level language, and Dena – a novel runtime tool, that work in concert to allow analysts to interact with a running system by hypothesizing about expected system behavior and posing queries about the system status. Using the given hypothesis and monitoring data, Dena applies statistical models to evaluate whether the system complies with the analyst’s expectations. The degree of fit is reported to the analyst as confidence scores. We evaluate our approach on a multi-tier dynamic content web server consisting of a Apache/PHP web server, a MySQL database using storage hosted by a virtual storage system called Akash and find that Dena can quickly validate analyst’s hypotheses and helps to accurately diagnose system misbehavior.
Figure 3.8: **Timing of Hypothesis Execution:** We measure the time to execute an expectation and notice that the bulk of the cost is fetching the data from the database while the time needed to perform statistical regression is small.
Chapter 4

Stage-Aware Anomaly Detection

In this chapter we introduce a real-time, low-overhead anomaly detection technique, which leverages the semantics of log statements, and the modular architecture of servers to pinpoint anomalies to specific portions of code.

4.1 Introduction

Operational logs capture high resolution information about a running system, such as, the internal execution flow of all individual requests and the contribution of each to the overall performance. Insights gained from operational logs have proved to be critical for finding configuration, program logic, or performance bugs in applications [109, 113, 112, 47, 80, 110].

Machine generated logging at each level of the software stack and at all independent nodes of a large-scale networked infrastructures create vast amounts of log data. This makes the main purpose for which logs were traditionally designed, i.e., processing by humans, intractable in practice.

To assist users to search for anomalous patterns in the large volume of logs, various automated data mining methods have been proposed [109, 110, 80]. The common feature of these methods is to apply text mining techniques, such as, regular expression matching to infer execution flows from the log messages and expose the anomalous execution flows to the user.

The conventional text-mining methods suffer from two problems: i) they rely on DEBUG-level logging which generates large volumes of data and makes compute and storage requirements excessive, and ii) thread interleaving and thread reuse hide the relationships between log messages and code structure.

The volume of diagnostic log data is large. The footprint of DEBUG-level logging, which is essential for diagnosis, is an order of magnitude larger than that of INFO-level logging. For instance, a Cassandra cluster of 10 machines, under a moderate workload, generates 500000 log messages per hour (5TB log data per day).
with DEBUG-level logging, a factor of about 2600 times more than with INFO-level logging. For this reason, text-mining approaches to anomaly diagnosis introduce significant costs for capturing, managing and analyzing the log messages. The common practice is to deploy a dedicated log collection and streaming infrastructure to store the logs, and to transfer them to a different (possibly remote) infrastructure for off-line analysis [15, 69, 30].

In addition to the overhead of conventional log mining methods based on DEBUG-logging, inferring the application’s semantics and execution flow from log messages is currently very challenging for two reasons: thread interleaving and thread reuse. Due to thread interleaving, log messages that belong to the same task do not appear in a contiguous order in the log file(s). The thread interleaving problem may be mitigated by printing the thread id with each log message, but it does not solve the second problem which is thread reuse. A thread might be reused for executing multiple tasks during its lifecycle. Inferring begin and end of individual tasks from the log messages generated by a single thread is difficult. Users often rely on ad-hoc complex rules to infer the boundaries of tasks from log messages.

In summary, currently users have only two choices: i) to forgo DEBUG-level logging in production systems and give up the crucial information that those logs provide for problem diagnosis, or ii) to pay the high costs of DEBUG-level logging in conjunction with ad-hoc, approximate, and error-prone log mining solutions [109, 80].

In this chapter, we introduce Stage-aware Anomaly Detection (SAAD), a low-overhead, real-time anomaly detection technique that allows users to benefit from low overhead logging, while still being able to access insights available only with detailed logging. SAAD targets the stage-oriented architecture commonly found in high-performance servers. Staged architectures execute requests in small code modules which we refer to as stages. SAAD tracks the execution flow within each stage by monitoring the calls made to the standard logging library. Stages that generate rare/unusual execution flow, or register unusually high duration for regular flows at run-time indicate anomalies.

Figure 4.1 illustrates the underlying principles of staged architectures. In this architectures, the server code is structured in stages, shown as the Foo, Bar and Baz blocks in the Figure. The code of any given stage, e.g., Bar, may be executed simultaneously by several tasks which are placed in a task queue at run-time to be executed by separate threads. Each task execution corresponds to a certain flow path taken in the execution of a given stage code by a thread, captured by the logging calls issued by that task.

SAAD leverages all log statements in the code (DEBUG- and INFO-level) as tracepoints to track task execution at run-time. It intercepts calls to the logger made by a task to record a summary of the task execution. SAAD ignores the content of the log messages, and does not write the log messages to disk. Upon a task completion, the summary of the task execution, which is a tiny data structure of few tens of bytes, is streamed to a statistical analyzer to detect anomalies in real-time.
Figure 4.1: High performance servers execute tasks in small code modules called stages. From the log statements encountered by each task at run-time, we can reason about the task execution flow and duration.

Since all tasks pertaining to a stage execute the same code, under normal conditions, each task exhibits statistically repeatable execution flow and duration. The statistical analyzer clusters the tasks based on their similarity to detect rare execution flow and/or unusually high duration.

We minimally instrument the code to insert a few tens of lines of code to delineate stages in the source code as runtime hints to track the start and termination of tasks. Since the number of stages is limited, the code modification is minimal compared to the magnitude of servers code with tens or hundreds of thousands of lines of code.

The contributions of our SAAD technique described in this chapter follow:

- **Limiting the search space for root causes:** SAAD leverages the architecture of server codes to limit the search space for root causes, by pinpointing to specific anomalous code stages in the code.

- **Detecting anomalies in real-time:** In contrast to existing log analytics that requires expensive offline text-mining, SAAD detects anomalies in real-time with negligible computing and storage resources.

- **Leveraging log statements as tracepoints:** SAAD uses the existing log statements in the code as tracepoints to track execution flows. It allows users to access insights available only at DEBUG-level logging at the same overhead as with INFO-level logging.
4.2 Motivating Example

We illustrate the stage construct of servers through a real-world example. We then illustrate how the log statements can be leveraged to detect anomalies in tasks.

4.2.1 HDFS Data Node Write

We illustrate the concepts of stage and task in the context of a write operation in the Hadoop Distributed File System (HDFS).
Figure 4.2 shows the execution of a write operation in the Hadoop Distributed File System (HDFS). In HDFS, data resides on a cluster of Data Node servers with 3-way replication for each data block. On each Data Node, the execution of the write request is handled by processing in two stages: DataXceiver and PacketResponder.

A task executing the DataXceiver (D) stage receives a packet, pushes it to the Data Node downstream and writes the packet in the local buffer. Another task executing the PacketResponder (P) stage sends acknowledgment packets upstream. Each task runs in a dedicated thread, and on each of the three Data Nodes, the same DataXceiver (D) and PacketResponder (P) stage might be executed in parallel for the potentially many client write requests that are executing concurrently.

Since many threads may execute the same stage (D and/or P) on the same node, as well as on different nodes, the server architecture just presented offers us opportunities for statistical analysis of many similar task execution flows for detecting outliers per stage, both within each node and across a server cluster.

Our key idea for capturing task execution flow and statistical classification is to track the calls made to the logger from the log points encountered during the execution of a task, as we describe next.

Detecting Anomalies. The intuition behind SAAD is that normal tasks that are instances of the same stage may register several different execution flows and/or variability in their duration as part of normal executions e.g., due to being invoked with different input parameters. But, they are expected to show repeatability in their execution flows.

We showcase the key ideas of our real-time statistical analysis on a simplified example of the DataXceiver stage on a Data Node in the HDFS write operation in Figure 4.3.

For the purposes of this example, without loss of generality, we show simplified log patterns of the tasks in this stage in Figure 4.4. We see that the usual log pattern of the tasks executing this stage, \([L_1, L_2, L_4, L_5]\), occurs 99\% of the time with a duration of 10ms. We see that the different log sequence \([L_1, L_2, L_3, L_4, L_5]\) occurs only 0.1\% of the time, due to the reporting of an empty packet abnormality (associated with \(L_3\)). Furthermore, we know that the overall performance of the client write operation depends on the performance of each individual task. The time difference between the beginning of a task and the last logging point it encounters is a good indicator for the duration of the task. For example, from Figure 4.3 we see that 0.9\% of tasks executing DataXceiver stage have 20ms duration, double the duration of normal tasks. Based on this example, we can see the opportunity to detect tasks with different execution flow and high duration in the total executions of this stage.

4.3 Design

In the following section, we present a high level overview of our Stage-Aware Anomaly Detection (SAAD) system. We then present a more detailed description of all components.


4.3.1 SAAD Design Overview

Stage-aware Anomaly Detection (SAAD) is comprised of two main components: a task execution tracker and a statistical analyzer as represented, at a high level, in Figure 4.5.

A task execution tracker running on each node of the server cluster intercepts the execution flow of tasks by
registering calls to the logging library per task, and produces a task synopsis at task termination. Task synopses are then tagged with semantic information, such as the stage that the task pertains to, and streamed out to a centralized statistical analyzer.

The centralized statistical analyzer periodically inspects tasks for outliers. We build an outlier model based on a trace of task synopses when the system operates without any known fault. Outlier tasks detected by the outlier model are the tasks with rare or new execution flows or the tasks with normal execution flow but with higher duration than normal. For a stage, if proportion of the outlier tasks to the normal tasks statistically exceeds the learned threshold in the outlier model, we consider that stage anomalous.

In SAAD, the capturing and streaming of task synopses, and eventually the anomaly detection are done in-memory, without any need to store the task synopses on persistent storage.

While, in our system, statistical analysis can be done independently, per-node, we chose to perform statistical analysis in a centralized fashion, for a cluster of nodes. Our choice is justified by the fact that transferring and processing task synopses are very lightweight, hence pose no scalability problem. On the up side, centralized processing of cluster-wide synopses increases statistical significance for the collected information faster, hence speeds up the process of building the anomaly detection model.

In the following, we describe the task execution tracker and the statistical analyzer.

### 4.3.2 Task Execution Tracker

The task execution tracker is a thin software layer that sits between the server code and the standard logging library. It tracks the execution flow of each task from the calls it makes to the logging library, and produces a summary of its execution. The task execution tracker i) identifies tasks at runtime, and ii) tracks execution flow of the tasks.
Identifying tasks

In stage architectures, a task is a runtime instance of a stage that is executed by a thread. We instrument the beginning of each stage code to track association of tasks and threads.

The beginning of a stage in the code is a location where threads start executing new tasks. These locations are identified from the two standard staging models as shown in Figure 4.6: i) Producer-Consumer model and ii) Dispatcher-Worker model. In the Producer-Consumer model, threads in the producer stage place requests in a queue, and threads in the consumer stage take the request for further processing. The threads in the consumer stage run in an infinite loop of dequeuing a request and executing it. Each request in the queue is handled by a consumer thread which represents a unique task. The Hadoop RPC library and the Apache Thrift library (used in Cassandra) adopt this model. In this model the beginning point of a consumer stage is the place where threads dequeue requests.

In the Dispatcher-Worker model, a thread in the dispatcher stage spawns a thread in the worker stage and delegates a task to it. This model is used in cases where a thread defers computation, e.g., by making an asynchronous call or parallelizing an operation. For instance, the HDFS Data Node DataXceiver in our motivating example uses this model. The beginning point of a worker stage is located at an entry point where threads start executing the code.

Execution tracking

During a static pre-processing pass over all server source code, we assign unique identifiers to all log points, augment each log statement with passing as argument its corresponding log identifier, and record the log point to log template associations in a log template dictionary.

Then, at runtime, the task execution tracker intercepts the calls to the logger, and registers the succession of log point identifiers encountered during the execution of each task. Each log point is represented by the unique position in a log point vector given by its pre-assigned log point identifier. The tracker further registers the frequency of each log point encountered by a task. Each log point encounter is accumulated in the corresponding entry in the log vector maintained within a per-task, in-memory data structure.
When a task terminates, the task execution tracker generates the synopsis of the task execution. This synopsis is on the order of a few tens of bytes and contains the stage identifier, the task unique id, its start time, and duration, and the frequency of each of the log points.

### 4.3.3 Stage-aware Statistical Analyzer

The statistical analyzer detects anomalies from the stream of task synopses at runtime. It detects anomalies that manifest in the form of an increase in outlier tasks in a stage. An outlier is a rare/new execution flow and/or an execution flow with unusually high duration.

Anomalies are reported in a human-understandable way to the user through our visualization tool. The visualization matches outlier tasks with names of stages and with the semantics associated with the information in log points encountered during execution.

In the rest of this section, we illustrate three steps of the statistical analyzer:

1) **Feature Creation.** From the task synopsis, we first create feature vectors for each task synopsis. We choose features that capture execution flow and performance aspects of a task.

2) **Outlier Detection.** Next, we construct a classifier to label tasks as outlier or normal. During runtime, the classifier is used to detect outlier tasks.

3) **Anomaly Detection.** We apply statistical tests to detect if the proportion of the outlier tasks exceeds a threshold. The outcome of the anomaly detection is presented to users for root-cause analysis.

#### Feature Creation

We extract two features capturing each task’s logical and performance behavior. For capturing logical behavior, we create a signature of the task’s execution flow from the distinct log points that it has encountered during execution. For the performance feature of a task, we use the task’s duration. We describe each feature in more detail next.

**Task Signature.** A task signature is a set of unique log points encountered by the task. Each log point in the signature indicates that the task has encountered the log point at least once. For example, the task signature for the normal task in Figure 4.4 is \{L1,L2,L4,L5\}. The slightest difference in signature is a strong indicator of a difference in the execution flow. More precisely, when signatures of two tasks are different in one or more log points, it means that one of them has executed a part of the code where the other one has not.

**Duration.** Duration is the time difference between the beginning of a task and the timestamp of the last log point encountered by the task. The duration of a task is a strong indicator of the task performance. Faults, such as a slow I/O request, that impact the system’s overall performance cause an increase in the execution time of tasks, which is visible in the increased timespans between log points of a task.
The outcome of feature creation is a feature vector for each task: \(<id, \text{stage}, \text{signature}, \text{duration}>\) which contains the task unique id, stage, signature, and duration of the task. The feature vector is used for outlier detection and anomaly detection, which we explain next.

**Outlier Detection**

For each stage, we classify tasks into normal and outlier based on the task signature and duration. We construct the classifier from a trace of task synopses when the system operates without any known fault.

For each stage, tasks are grouped based on their signatures. We count the number of tasks per signature, and determine the percentile rank of each signature in descending order. Signatures with rank higher than a threshold are considered *logical outliers*. For example, by setting the threshold at the 99th percentile, signatures that account for less than 1% of tasks are considered outliers. The number of signatures are expected to be finite and relatively small because of the finite number of execution flows when the system runs normally. In fact, normal execution flows account for the vast majority of the tasks. We observed this in the systems we studied as shown in Figure 4.7 where 20% of the signatures account for more than 95% of the tasks in HDFS, HBase, and Cassandra.

Finally, we group tasks with the same stage and signature. For each group, we compute the 99th percentile of the tasks’ duration as the performance outlier threshold. The tasks with duration greater than the threshold are considered *performance outliers*.

![Figure 4.7: Distribution of signatures.](image)

Most of the tasks follow a few execution paths. In HDFS Data Node, 6 out of 29, in HBase, 12 out of 72 and in Cassandra, 10 out of 68, of signatures account for 95% of all tasks.

The duration of tasks with certain execution flows do not register a skewed distribution, which is a prerequisite for being able to determine and select a meaningful outlier threshold. As a result, we cannot accurately classify these tasks into performance outlier. To detect these execution flows, we apply a standard k-fold cross-validation technique. For each signature, we divide the training trace into k equally sized subsets. We construct the outlier classifier from the k-1 subsets and measure the percentage of performance outliers. We repeat the same process for the remaining k-1 subsets. If the average of the percentage of performance outliers for all k sets is significantly higher than the predefined outlier threshold, we discard the respective signature for the purpose of performance outlier detection.
Anomaly Detection

We define an anomaly as a statistically significant increase of outlier tasks per stage. To detect an anomaly, we periodically run statistical tests to verify whether the proportion of outlier tasks exceeds a threshold. We refer to an increase in logical outliers as a logical anomaly, and an increase in performance outliers as a performance anomaly.

In the following, we illustrate the logical and performance anomaly detection in more detail.

**Logical anomaly.** A stage has a logical anomaly if at least one of two conditions is fulfilled: i) using a t-test with significance level of 0.001, the following hypothesis is rejected: the proportion of logical outliers is less than or equal to the proportion of logical outliers observed in the training data, or ii) we observe a new signature that we have not seen during training. A new signature indicates a new execution flow, which can be a strong indication of a fault. For instance, consider a fault that causes a task to terminate prematurely. The premature termination prevents the task to hit some of the log points it would normally encounter, and results in a signature that would not been seen in the absence of the fault.

**Performance Anomaly.** For each stage, we group tasks per signature and calculate the proportion of performance outliers. We use a t-test with significance level of 0.001 to verify the following hypothesis: the proportion of performance outliers is less than or equal to the proportion of performance outliers of that signature in the training data. If the hypothesis is rejected, the stage has a performance anomaly.

**Anomaly Reporting.** We present the detected logical and performance anomalies in a human understandable fashion for the users and developers to inspect. Each anomalous signature is presented to the user by its stage name, and the list of log templates of its log points. Log templates contain the static portions of the log statements in the code, which reveals the semantics of the execution flow.

4.4 Implementation

4.4.1 Task Execution Tracker

We modified log4j and added the task execution tracker as a thin layer between the server code and the log4j library. log4j is the de facto logging library used by most Java-based servers including Cassandra, HBase and HDFS. Our task execution tracker consists of about 50 lines of code.

Tracking task execution records function calls to the logging library from the log statements by each task. We instrument the beginning of each stage with an explicit stage delimiter instruction: setContext(int stageId) which hints to the task execution tracker that the thread is about to execute a new task, and passes the stage id. When the setContext(int stageId) function is invoked by a thread, the tracker creates a
data structure in the thread local storage representing the task. It populates the data structure with stage id, a unique id and the current timestamp. The data structure is also initialized with a map data structure that is used to maintain the ids and frequency of log points that will be visited by the task. For every log point encountered by the thread, the map is updated: if the log point is visited for the first time, an entry of log point id will be initialized with value 1 and will be added to the map, otherwise the value associated with log point id will be incremented by 1.

**Synopsis.** When a task terminates, the tracker generates the execution synopsis of the task from the thread-local data structure. The synopsis is a semi-structured record with the following fields:

```c
struct synopsis{
    byte sid; //stage id
    int uid; //unique id per task
    int ts; //task start time (ms)
    int duration; //task duration (us)
    struct {
        short int lpid;//log point id
        int count;//frequency of visit
    }log_points[];
}
```

Synopses are storage efficient. Since they are an order of magnitude smaller than the size of actual log messages, by storing synopses, we substantially reduce the volume of monitoring data. For instance, if each log message is 120 characters and the number of log messages generated by a task is 5, it takes 600 bytes to store the log messages generated by the task. Instead, a task synopsis takes at most 33 bytes. This represents an 18 fold reduction in space.

**Determining Task Termination.** While the beginning point of each stage is unique and can be identified from the source code, the exit points are multiple and hard to reason about statically from the source code. As an analogy, the beginning point of a function in most programming languages can be easily identified, but the function may have multiple exit points, e.g., `return` statements, or even exceptions which are unknown until runtime. For this reason, the termination of tasks must be inferred at runtime by the tracker. In the case of the producer-consumer model, we infer the termination of a task when the thread is about to start a new task. If a task synopsis data structure is already initialized in thread private storage, it indicates that the thread is finished with the previous task. In the case of the dispatcher-worker model, the termination of a task is inferred from the termination of the worker threads. In Java, we infer termination of a thread through the garbage collection mechanism. When a thread terminates, objects allocated in its private storage become available for garbage collection. Before the garbage collector reclaims space for an object, it calls a special method named `finalize()`. We add proper instructions in this method to generate the task synopsis.
Instrumentation

Stages. We wrote a script about 40 lines of Ruby code to parse and analyze the Java source code to identify the beginning of stages to add the proper instrumentation. In most cases, the beginning of a stage code corresponds to the place a thread starts executing a code which is the `void public run()` method of `Runnable` objects. Instrumenting this place covers: i) all cases of “dispatcher-worker”, where a thread is instantiated and a task is delegated to it, and ii) cases of “producer-consumer”, where the consumer stage is implemented as a standard Java managed thread-pool construct called `Executor` that accept tasks in from of `Runnable` objects in the input queue.

For other cases of “producer-consumer” that are not based on Executors, we manually add the instrumentation to places in the code where a thread begins a new stage by reading its next request from a request queue. Since in most cases, Java applications use standard queuing data structures, our script identifies and presents dequeuing points in the source code for manual inspection.

Although this is a one-time procedure, it is not labor intensive because the number of stages is limited. There are 55 stages in HDFS, 38 in HBase Regionservers, and 78 in Cassandra.

Log points. We also instrument log statements to pass an id to identify their location in the code at runtime. We wrote a Ruby script (of about 50 lines of code) which parses the source code and identifies the log statements, and rewrites the log statement with a unique log id, for instance, a `log.info(...)` statement is replaced with `log.info(uid,...)`, where the uid is a unique identifier of the log point.

It is common practice that developers add an if statement to check the configured DEBUG-level logging right before the debug and trace log statements to avoid the unnecessary overhead of producing and sending the log message to the logger:

```java
if(isDebugEnabled())
  log.debug(...)
```

In such cases, our script adds the log statement uid as an input to the statement that checks for verbosity:

```java
if(isDebugEnabled(uid))
  log.debug(...)
```

Our script successfully instrumented 3000+ log statements in HBase, HDFS, and Cassandra in less than one minute. The script also builds a dictionary of log templates, i.e., log statements and the information of their respective place in the source code. The log template dictionary is only used for visualization and provided to the user for the purpose of manual root cause diagnosis.
4.4.2 Statistical Analyzer

We developed the statistical analyzer in R [19]. R is a scripting language with versatile statistical analysis packages. Although it is not designed for high performance computing and it is not multi-threaded, it never became a bottleneck in any of our experiments to achieve real-time anomaly detection. Our implementation handles streams of task synopses as fast as they are generated, up to the maximum we observed in our experiments which is 1500 task synopses per second. Constructing the statistical model is also efficient: it takes about 60 seconds per host for a 1 hour trace of data including 5.5 million task synopses. The efficient model building confirms our design decision to limit the computation for training to counting and computing percentiles. During runtime, the computation is extremely light-weight – limited to hash-map operations to determine if the task’s signatures belong to the logical outlier set, simple floating point comparison to determine if the duration falls in the performance outlier region, and t-tests for detecting logical and performance anomalies. The synopses are temporarily buffered in memory during model construction. In our experiments, we never exceeded 500MB memory demand to buffer synopses during the model construction.

4.5 Experiments

We evaluate our Stage-aware Anomaly Detection (SAAD) framework on three distributed storage systems: HBase, Hadoop File System (HDFS) and Cassandra. These systems are among the most widely used technologies in the “big data” ecosystem. They consist of distributed components and generate a large volume of operational logs. To avoid generating large volumes of log data, in production environment, the common practice is to set the logging to INFO-level logging. However, by setting the logging to INFO-level logging, we lose valuable information about the execution flows in the system. In this section we show that with SAAD we keep the generated logs at INFO-level, while taking advantage of execution flows recorded in task synopses.

We demonstrate that SAAD is effective in pinpointing anomalies in real-time with minimal overhead. The evaluation begins with measuring SAAD’s overhead (Section 4.5.3) with respect to i) task execution tracker runtime, ii) volume of generated monitoring data (task synopsis), and iii) the statistical analyzer computing resource requirements for real-time anomaly detection.

Then, in Section 4.5.4 and Section 4.5.5 we evaluate SAAD effectiveness in detecting anomalies on HBase/HDFS and Cassandra. We demonstrate that SAAD narrows down the root-cause diagnosis search by detecting the stages affected by injected faults. In Section 4.5.4 we highlight the advantage of SAAD in uncovering masked anomalies that lead to system crash or chronic performance degradation with no explicit error/warning log messages. In Section 4.5.5 we describe our experience with SAAD in revealing some real-world bugs in HBase and HDFS.

Finally, we show the accuracy of SAAD through a comprehensive false positive analysis in Section 4.5.6.
Before we present the experiments, we begin with a high-level background on HBase, HDFS, and Cassandra to help the reader understand the results we present in this section.

### 4.5.1 Testbed

In this section, we provide the background on HBase, HDFS and Cassandra that we deem necessary to understand and interpret the results. HBase, HDFS, and Cassandra are open-source and written in Java.

**Figure 4.8: Overview of HBase/HDFS.**

**HBase/HDFS.** HBase is a columnar key/value store, modeled after Google’s BigTable [35]. HBase runs on top of Hadoop Distributed File System (HDFS) as its storage tier. Figure 4.8 shows the architecture of HBase/HDFS.

HBase horizontally partitions data into regions, and manages each group of regions by a Regionserver. A master node monitors the Regionservers and makes load balancing and region allocation decisions. HBase relies on Zookeeper [60] for distributed coordination and metadata management.

HDFS provides a fault-tolerant file system over commodity servers. On each server, a Data Node manages a set of data blocks and uses the local file system as the storage medium. HDFS has a central metadata server called Name Node. It holds placement information of blocks and provides a unified namespace.

**Cassandra.** Cassandra is a peer-to-peer distributed system, and unlike HBase and HDFS, it does not have a single metadata/master server. Its distributed data placement and replication mechanism is based on peer-to-peer principles akin to Distributed HashTables (DHT) [100] and Amazon Dynamo [45]. Cassandra relies on the local file system as storage medium.
Chapter 4. Stage-Aware Anomaly Detection

Figure 4.9: Storage Layout of HBase and Cassandra.

Storage Layout of HBase and Cassandra. Figure 4.9 shows the storage layout of Cassandra and HBase. The storage layout is based on Log-Structured Merge Tree [83]. In this layout, writes are applied to an in-memory sorted linked-list called MemTables/MemStores, for efficient updates. Once a MemTable grows to a certain size, it is flushed to the disk and stored in a sorted indexed file called SSTable. To guarantee persistence, each update is appended to a write-ahead-log (WAL) and synced to the file system. When the MemTable is flushed, the entries in the WAL are trimmed. For reads, the MemTables are searched for the specified key. If not found, SSTables are searched on disk in reverse chronological order, i.e., the newest SSTables are searched first. Flushing a MemTable and merging it to stored SSTables is called minor compaction. As the number of SSTables grows beyond a threshold, they are merged into fewer SSTables in a process called major compaction.

4.5.2 Experimental Setup

We ran our experiments on a cluster of HBase (ver. 0.92.1) running on HDFS (ver. 1.0.3), and Cassandra (ver. 0.8.10). Our testbed cluster consists of nodes with two Hyper-Threaded Intel XEON 3Ghz processors and 3GB of memory, on each node. For the HBase setup, each server hosts an instance of a Data Node and a Regionserver. The HBase Master, HDFS Namenode, and an instance of Zookeeper are all collocated on a separate dedicated host with 8GB RAM. For the Cassandra setup, each Cassandra node runs on a single server.

Workload Generator. To drive the experiments, we use Yahoo! Cloud Serving Benchmark [42]. YCSB (ver. 0.1.4) configured with 100 emulated clients. YCSB is an accepted workload generator for benchmarking NoSql key/value databases, including HBase and Cassandra. YCSB generates requests similar to real-world workloads.

Workload. In practice, most key/value databases such as Cassandra and HBase reside below several layers of caching. Therefore, read requests are mostly absorbed by the caching tiers before hitting the database. Hence,
most requests that reach the Cassandra and HBase tiers are write operations. We chose a write-intensive workload mix for our experiments, to make the workload mix resemble the kind of mix that these database systems handle in practice.

### 4.5.3 Overhead

**Overhead of Task Execution Tracking**

In this section, we measure the runtime overhead of the task execution tracker in terms of its effect on the performance of the application and its memory footprint.

**Performance Overhead.**

We compare the throughput of the subject system with SAAD (i.e., modified log4j library and the execution tracker), to the original system. The logging level for both cases is set to INFO-level, which is the default configuration in production systems.

Figure 4.10 compares the throughput of HBase and Cassandra, with and without SAAD. We see that the throughput of the system with and without SAAD is not significantly different. This demonstrates that SAAD imposes insignificant overhead on the system.

![Figure 4.10: SAAD Overhead. Normalized average throughput of HBase and Cassandra with SAAD is compared to their original versions (without SAAD). The error bars (normalized) indicate the variation of the measured throughput every 10 seconds.](image)

**Memory Overhead.**

We evaluate the memory footprint of the task execution tracker. The tracker buffers the task synopses (timestamp, stage id, unique id, log frequency vector and the duration), which are 48 bytes on average. Once a task is terminated, its synopsis is sent to the statistical analyzer. In our experiments, the memory usage of the task execution tracker during runtime always remained under a few kilobytes.
Storage Overhead

In this section, we show the effectiveness of SAAD in reducing the storage overhead of monitoring data storage. We first quantitatively show that DEBUG-level logging provides more insight into the system in terms of distinct execution flows. Then, we show that the storage overhead of SAAD is an order of magnitude less than the storage demand for storing log messages.

In Figure 4.11, we see that logging at DEBUG-level reveals more distinct execution flows than INFO-level logging. For instance, INFO-level logs in HBase exposes no better than 30 unique execution flows, 40% less than the total unique execution flows exposed by DEBUG-level logging.

However, the number of log messages generated at DEBUG-level logging is substantially more than the number of log messages generated at INFO-level logging. For instance, in a one hour run, Cassandra generates 11.9 million log messages at DEBUG-level logging, while at INFO-level, it generates 4500 log messages; a 2600 fold difference. This means that storing and processing DEBUG-level logs requires an order of magnitude more storage demand than INFO-level logs. Due to this fact, servers are configured to INFO-level logging in production environments during normal operation.

Storage overhead of SAAD. In Figure 4.12 we compare the volume of logs generated in DEBUG mode with the volume of synopses generated for HDFS, HBase, and Cassandra. We see that volume of task synopses are between 15 to 900 times less than log messages. This highlights the strength of our approach in reducing the storage overhead and processing time, which is a major contributing factor to our real-time anomaly detection.

Statistical Analyzer Overhead

We evaluate the statistical analyzer overhead in terms of number of CPU cores needed to process task synopses in real-time. Our current implementation of the statistical analyzer runs on one core.

For comparison, we focus on the most resource-intensive phase of conventional log analytics methods, that is text-mining. These techniques [109][112] reverse match log messages to the log statements in the code. Based on
static code analysis, these methods generate a regular expression for each log statement that matches all possible log messages that the log statement would produce. They use these regular expressions to reverse match the log messages to their corresponding log statement in the code. Reverse matching over large volumes of data is time-consuming. To speed up the reverse matching process, Xu et al. use MapReduce framework.

In order to show that our statistical analyzer has significantly lower overhead than state-of-the-art anomaly detection methods based on log mining, we implemented a MapReduce job similar to the one used by Xu et al. [109]. The MapReduce job processed one hour of log data of a Cassandra cluster with 11.9 million log messages (about 1.6GB). It took about 12 minutes of batch-processing on a dedicated cluster of 8 cores to reverse match the log data.

SAAD, on the other hand, by circumventing text parsing through tracking of log points and generating synopses on the fly, requires only one core to produce similar results in real-time.

### 4.5.4 Cassandra

In this section, we evaluate SAAD in detecting and pinpointing anomalies in a Cassandra cluster. We show that SAAD can detect the problems that common log monitoring systems which search for error/warning messages are unable to detect.

We conducted several experiments to thoroughly evaluate SAAD on various I/O faults of a Cassandra node. In each experiment, a different fault is injected on only one node, to emulate partial failures which are hard to detect due to fault masking.

**Failure Model.** We injected 8 different faults based on the following factors:

- **I/O activity.** Cassandra has two major types of I/O activities related to MemTables and write-ahead-log (WAL) (see section 4.5.1).
• **Failure mode.** We consider two fault modes: error and delay faults. An error fault explicitly results in a returned error code, emulating an I/O failure. For example, when writing MemTables to disk, some of the operations receive an error message. A delay fault causes the I/O to pause for 100ms. We inject faults (error and delay) through Systemtap [17].

• **I/O operations.** We injected faults on read or write operations.

Table 4.1 shows the experiments and the description of faults.

In each experiment, each fault is injected at two intensity levels, low and high intensity. The intensity is controlled by the probability of random I/O requests that are subjected to the fault. A low intensity fault affects 1% of I/O requests and a high intensity fault affects 100% of the I/O requests. The duration of each experiment is 60 minutes. First we inject a low-intensity fault at minute 10 for a period of 10 minutes (until minute 20). Then, at minute 30, we inject a high-intensity fault for a period of 10 minutes (until minute 40).

For each experiment, we show the performance and logical anomalies per stage to highlight the ability of SAAD to detect the relevance of anomalies to the fault. As a baseline, we compare our method with common log monitoring alert systems, where the logging library alerts the user when an error log statement is generated.

In these experiments, we will show that the anomalies that our method captures are a superset of the anomalies that conventional error alert systems report, in addition to the benefit of associating the stage name and the execution flows in terms of log statements.

In these experiments, the statistical model is constructed from a 2 hour trace with about 21.7 million task synopses. The model construction (training) took about a minute for each host (4 minutes in total).

Table 4.1: **Fault Description.** We injected eight faults based on combinations of I/O activities, type of failures, and I/O operations. Delay faults induce 100ms delay to the target I/O requests. An error fault is induced by intercepting the target I/O requests issued by Cassandra and returning an I/O error code. Each fault is injected at two levels of intensity determined by the probability that an I/O request is affected: 1% for low intensity and 100% for high intensity.

<table>
<thead>
<tr>
<th>I/O Activity</th>
<th>Mode</th>
<th>I/O Operation</th>
<th>Description</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>WAL</td>
<td>Error</td>
<td>Write</td>
<td>Error on write operation to WAL (write-ahead-log)</td>
<td>Figure 4.13(a)</td>
</tr>
<tr>
<td>WAL</td>
<td>Delay</td>
<td>Write</td>
<td>Delay on write operation to WAL</td>
<td>Figure 4.13(b)</td>
</tr>
<tr>
<td>MemTable</td>
<td>Error</td>
<td>Write</td>
<td>Error on write operation during flushing a MemTable to an SSTable</td>
<td>Figure 4.13(c)</td>
</tr>
<tr>
<td>MemTable</td>
<td>Delay</td>
<td>Write</td>
<td>Delay on write operation when flushing a MemTable to an SSTable</td>
<td>Figure 4.13(d)</td>
</tr>
<tr>
<td>MemTable</td>
<td>Error</td>
<td>Read</td>
<td>Error on read operation from SSTables</td>
<td>Figure 4.14(a)</td>
</tr>
<tr>
<td>MemTable</td>
<td>Delay</td>
<td>Read</td>
<td>Delay on read operation from SSTables</td>
<td>Figure 4.14(b)</td>
</tr>
<tr>
<td>WAL</td>
<td>Error</td>
<td>Read</td>
<td>Error on read operation from WAL</td>
<td>Figure 4.14(c)</td>
</tr>
<tr>
<td>WAL</td>
<td>Delay</td>
<td>Read</td>
<td>Delay on read operation from WAL</td>
<td>Figure 4.14(d)</td>
</tr>
</tbody>
</table>

**WAL-error-write: Error on writing to write-ahead-log.** In this experiment, write operations to WAL by the Cassandra node on host 4 are intercepted, and replaced with an error return code. Figure 4.13(a) shows the results.
Despite the I/O failure, the system masks the failure. It generates only one error log message at minute 18, and the overall throughput stays unaffected until minute 32, long after the fault is injected at minute 10.

SAAD uncovers several logical anomalies in stage Table. The anomalous task signatures in this stage indicate that MemTables are frozen, meaning that this Cassandra node stops applying writes to MemTables.

Updating memtable requires appending an entry to WAL, this signature hints that write operations to write-ahead-log are failing.

This anomaly does not result in an error log message; hence, it could not be detected by searching for error messages in log data. Even worse, no single log message points to the anomaly. This anomaly can only be detected from rare execution flow.

Table 4.2 shows the signatures of a normal execution flow and the detected anomalous execution flow. We see that the normal execution flow includes four log statements, while the anomalous execution flow consists of only
the first statement. The log statement reporting MemTable being frozen appears in both normal and rare execution flows. One would expect that this log statement is an error. But, in fact it is a normal behavior. This log statement means that a task must momentarily wait until a lock is released before it can proceed with mutating a MemTable. However, the injected fault on write to WAL causes a silent failure. The failure prevents new appends to be applied to the WAL, and consequently, causes a task that is in the middle of applying mutation and appending to WAL to get indefinitely stuck and never release the lock it hold on MemTable. As a result, other tasks cannot proceed in mutating the MemTables, and terminate prematurely. This premature termination of the tasks is reflected as a rare execution flow.

This highlights the strength of SAAD, which uncovers anomalies from execution flows inferred from existing log statements compared to conventional log monitoring methods which watch for certain error messages such as warnings and/or errors.
Table 4.2: Signature of a normal execution flow and the signature of the anomalous execution flow that indicate MemTable is frozen. This anomaly can only be detected as a rare execution flow.

<table>
<thead>
<tr>
<th>Description of log statements</th>
<th>Normal</th>
<th>Anomalous</th>
</tr>
</thead>
<tbody>
<tr>
<td>MemTable is already frozen; another thread must be flushing it</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Start applying update to MemTable</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Applying mutation of row</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Applied mutation. Sending response</td>
<td>✔️</td>
<td>✔️</td>
</tr>
</tbody>
</table>

The effect of the frozen MemTables on host 4, in which the fault was injected, eventually appears on hosts 1 and 2, which is uncovered by SAAD in the stage WorkerProcess. Since appending to the WAL and updating MemTables are the mandatory conditions to complete a write, the Cassandra node on host 4 never completes any of the writes that it receives. Once other Cassandra nodes notice that host 4 has become non-responsive, they start delegating writes to random healthy nodes, and request those healthy nodes to retry sending the writes to the failed node (host 4) at a later time. This process of delegation to random nodes for a later retry is called “hinted hand-off”. The anomalous logical signatures detected on hosts 1 and 2 indicate that “hinted hand-off” writes for host 4 are timing out.

Eventually as writes are indefinitely buffered in memory on host 4, the effect of memory pressure becomes visible as a dozen of error messages at minute 44, and shortly after that, the Cassandra process on host 4 crashes.

**WAL-delay-write**: Delay on writing to write-ahead-log. This fault delays write operations to WAL on the target node (host 4). Figure 4.13(b) shows the results of this experiment. SAAD detects several performance anomalies at the WorkerProcess and StorageProxy stages on host 4 during the high-intensity fault. The WorkerProcess stage holds worker threads that handle incoming requests from clients. The signature of the outlier tasks in this stage reveals that execution flows associated with applying mutation to rows in MemTables are slowed down. The performance anomalous signatures in StorageProxy also indicate slowdown in applying mutations to WAL.

Since mutating MemTable and adding an entry to WAL are done transactionally, from these two signatures the user can reason that a slow down on write to WAL is the cause of the problem.

**MemTable-error-write**: Error on write when flushing MemTables. This fault affects minor compaction operations in which MemTables are flushed to disk. Figure 4.13(c) shows the results of this experiment. SAAD detects logical anomalies in the MemTable stage that serializes MemTables and flushes them to disk. Also, we see anomalies in the CompactionManager stage. This stage merges several SSTables into one. To do so, it reads them to memory, merges them into one MemTable, and then writes the MemTable to disk as a new SSTable. Anomalies in CompactionManager and MemTable hint to I/O problem of writing to SSTables.

During the low intensity fault, throughput does not degrade, because only 1% of compaction operations fail. But, during the high-intensity fault, the effect of the fault gradually becomes visible in the performance. During
this period, since Cassandra cannot flush the MemTables, memory pressure escalates. It is detected in the garbage
collection stage GCInspector shortly after the high-intensity fault is in effect. We see that even after the fault
is lifted, at minute 40, the lingering effect of memory pressure is detected in the garbage collection stage.

**MemTable-delay-write: Delay on write when flushing MemTable.** This fault slows down minor compaction
operations and, as result of that, the affected node slows down in applying updates. Figure 4.13(d) shows the re-
sults of this experiment. SAAD detects consecutive performance anomalies at CommitLog, LocalReadRunnable
and WorkerProcess stages. Since the CommitLog stage trims WAL once a MemTable is successfully flushed
to disk, a slow-down in this stage hints at a slow-down in MemTable writing to disk. Slowdown in flushing
MemTables affects write operations which is detected as performance anomalies in WorkerProcess stage.
The signature of the anomalous tasks in this stage indicates that these tasks are performing write operations.

**MemTable-error-read: Error on read from SSTables.** Figure 4.14(a) shows the results of this experiment.
SAAD detects logical anomalies at the CompactionManager stage, which is responsible for merging several
SSTables into one. The merging operation involves reading SSTables. Since the Compaction occurs periodically,
i.e., when number of SSTables exceed a threshold, we observe only four logical anomalies in this stage during the
experiment.

**MemTable-delay-read: Delay on read from memtables.** As previously noted, Cassandra uses the LSM-tree
data structure to store and retrieve records. To read an item from the LSM-tree, first MemTables are searched. If
the item is not found, the SSTables on disk are searched in chronological order – newest to oldest. Hence, a read
operation can cause several read operations from disk. Figure 4.14(b) shows the results of slowing down reading
from SSTables. We see that SAAD detects several performance anomalies in LocalReadRunnable stages,
which serves read requests from local disk.

**Wal-error-read and Wal-delay-read : Error and delay on read from write-ahead-log.** Figure 4.14(a) shows
the results of this experiment. As expected, since write-ahead-logs are exclusively written to during normal
operations, injecting error and delay on reads causes no performance nor logical anomalies.

**Usecase: Uncovering Masked Failure**

Distributed systems such as HBase, HDFS, and Cassandra are designed to tolerate faults – they mask failures. If
these failures go undetected, they may eventually affect the overall performance. The effect of failures may appear
on the system’s key performance indicators several minutes after the occurrence of a fault. Hence, revealing
masked faults is helpful for root-cause analysis and even preventing a major failure such as a general outage in
the system.

In this section, we show that fault masking in Cassandra may lead to catastrophic consequences. Cassandra is
especially designed to work continuously without interruption under a wide range of faults, even under network
Figure 4.15: **Hazard of fault-masking.** Fault on host 4 renders the host read-only but throughput remains unaffected despite the write-intensive workload. Shortly after introducing the second fault on the host 3, throughput drops to zero. At this point, Cassandra nodes cannot maintain a three-way replication for any data. As a result, it stops accepting new writes.

partitioning and storage medium malfunction. We observed in that a fault on write operation to the WAL led to a silent failure: MemTables became frozen. Except for one error message shortly after the fault is injected, the Cassandra node reported no log messages to indicate that it had stopped accepting new updates. This anomaly did not show itself in the performance metrics either. Its throughput remained unaffected long after the fault occurred.

We repeated the WAL-error-write experiment, but this time we injected the faults on two nodes instead of one, both faults are low intensity faults with 1% of requests begging affected. The first fault occurs at minute 10 on host 4, and the second fault occurs at minute 30 on host 3, each lasts for 10 minutes. The results are shown in Figure 4.15. We see that, after the first fault that renders host 4 read-only (it stops accepting write requests), the throughput remains unaffected despite the write-intensive workload. Cassandra manages to serve writes from the remaining three nodes. Shortly after introducing the second fault, on host 3, throughput abruptly plummets to zero. At this point, Cassandra nodes cannot maintain a three-way replication scheme for any replicas. As a result, the Cassandra cluster stops accepting new writes.

In summary, after the first Cassandra node is affected, only one error message is generated at minute 20 with no indication that MemTables are frozen. The fault goes masked until the second fault is injected, and eventually halts the whole cluster.

### 4.5.5 HBase/HDFS

In this section, we evaluate SAAD in detecting anomalies in a cluster of HBase/HDFS servers. In this experiment, we injected faults over the course of 3 hours. We introduced the faults by launching one or more background processes of `dd /dev/urandom dummy count=1K count=1M` command on all hosts. This command
emulates a disk hog; it hogs the bandwidth of local disks. It also makes several system calls which raise many interrupts. Interrupts steal CPU cycles from the kernel processes and cause slowdown of other kernel activities, including network operations. In Table 4.3 we show the timeline of the injected faults. We began with a low intensity fault and gradually escalated the intensity in the subsequent faults.

Table 4.3: Description of injected faults. Faults are induced on all 4 hosts.

<table>
<thead>
<tr>
<th>Fault</th>
<th>Span</th>
<th># of dd processes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low-intensity</td>
<td>8-16</td>
<td>1</td>
</tr>
<tr>
<td>Medium-intensity</td>
<td>28-44</td>
<td>2</td>
</tr>
<tr>
<td>High-intensity-1</td>
<td>56-64</td>
<td>4</td>
</tr>
<tr>
<td>High-intensity-2</td>
<td>116-130</td>
<td>4</td>
</tr>
</tbody>
</table>

Fault Diagnosis

Figures 4.16(a) and 4.16(b) shows the detected anomalies, per stage, in HBase Regionservers and HDFS Data Nodes, respectively. In the figures, we overlay the results with error messages that appear in log messages.

Low-intensity fault: This fault begins at minute 8 and lasts for 8 minutes. We observe some performance anomalies on Regionserver 1 and Regionserver 2. Although the fault is injected on all hosts, we don’t observe performance anomalies on the other hosts. The reason is that during this period of time, Regionserver 1 and Regionserver 2 were receiving more operations than other Regionservers and were already overloaded. The lightweight hog impacted their performance. The hog did not affect any of the Data Nodes, which are expected to be disk bound. This suggests that the Data Nodes were not issuing disk operations, or the hog did not affect the disk bandwidth.

Medium intensity fault: This fault is induced between minutes 28 and 44. This fault is more intense than the first fault. With this fault, we observed that the performance outliers significantly increased on all Regionservers. Our model isolates the ‘get’ RPC calls in stage Call as anomalous signatures. This suggests that read operations are slowed down. Since we see no performance anomalies on Data Nodes, this pattern suggests that the slow-down is most likely caused by cpu contention, rather than I/O slow-down.

High intensity fault-1: This fault is injected between minutes 56 and 64. This fault was the most severe fault we injected into the system which led to a surprising fault scenario; Regionserver 3 unexpectedly crashed. As a result of this crash, we see a surge of logical outliers on all Regionservers and Data Nodes. Our further investigation uncovered that an unfixed bug in the HDFS client library was the cause of the crash, which we explain next. Bug: premature recovery termination. Our model isolated task signatures that indicated that Regionserver 3 engaged in a repetitive cycle of trying to send recovery requests to Data Nodes, while the requested Data Nodes had been already working on the recovery. This bug occurs when a Data Node is slow to recover a block. While the Data
Figure 4.16: Anomalies per stage in HBase Regionservers and HDFS Data Nodes.
Node indicates to the Regionserver that it is already in the middle of recovery, the Regionserver misinterprets the response as an exception and repeats the recovery request. The anomaly in block recovery is detected in the RecoverBlocks stage on Data Node 3 (Figure 4.16(b)).

In cases where the recovery is requested for a block that contains the Regionserver write-ahead-log data (HLog), the server stops processing any write requests (as a rule to guarantee persistence) until the recovery is confirmed on all the Data Nodes that hold a replica of the block. This repetitive cycle eventually leads to exceeding the number of allowed retries and causes the Regionserver to crash. In our case, the injected hog was the cause of Data Node slowdown. After the Regionserver crashes, the master assigns its regions to other Regionservers. The effect of this is observed as logical anomalies on other Regionservers as they engage in load-balancing. We made this diagnosis efficiently by only looking at a very small set of signatures, and their corresponding log templates. During this period, we observed a significant increase in performance anomalies on other Regionservers as well as Data Nodes.

**High intensity fault-2:** This fault is injected between minutes 116 and 130. Like the previous fault, the intensity of this fault was high. We were surprised that the increase in the outlier signals on most servers was not as severe as for the previous fault, i.e., high intensity fault 1 discussed above. We noticed that during this hog, unlike the previous ones, we had very few ‘log sync’ tasks on Regionservers which are in charge of flushing write-ahead-logs to HDFS. This suggests that few write operations occurred during this period of time, and in fact most performance anomalies were read operations, not write operations. After more investigation, we uncovered a hard-coded misconfiguration in the workload generator; YCSB emulator version 0.1.4. The YCSB configures its HBase client to batch ‘put’ operations on the client side and to periodically send them in one single RPC call. This artificially boosts performance of write operations, at the expense of delaying writes on the client side. The writes were persisted on Regionservers only after a significant lag of about 9 minutes on average. It must be noted that batching put operations violates the benchmark specifications.

**Major Compaction.** Close to the end of our experiment, we observed an unexpected spike of outliers on the Regionservers and Data Nodes. Our model detects logical outliers in the CompactionRequest and CompactionChecker stages on Regionservers. These stages are in charge of performing major compaction operations, where versions of key/values that stored in separated files (SSTables) are consolidated into fewer files. During this operation Regionservers issue many I/O requests to HDFS; therefore, we observe that on all Data Nodes performance and logical anomalies in DataXceiver stage is detected. The DataXceiver stage is responsible for handling write operations. This is a case of a false positive where a legitimate but rare activity is misidentified as an anomaly. In this case, our system could avoid the logical anomaly false positives as the result of major compaction, if the trace used to construct the outlier model had a case of major compaction.
Usecase: Uncovering Bugs and Misconfigurations

Our approach produces a white-box, descriptive model which enables users to inspect rare execution flows for potential software bugs and/or misconfigurations.

During our fault injection experiments, the system was under stress, which sometimes led to uncovering elusive bugs or subtle misconfiguration problems. We found two bugs (other than the “premature recovery termination” discussed in the previous section) and two misconfigurations. A brief description of each anomaly is shown in Table 4.4. In each case, our model identified the symptoms of the anomalies as a rare execution flow.

The descriptive nature of signatures provides the possibility of drilling down from high level anomaly description in form of log points in the source code to actual raw log records that are captured and stored by the standard logger. With the semantic information associated with the signatures, i.e., stage and log templates, we could diagnose the root-causes efficiently. In all cases, the log points isolated by our model closely matched the descriptions of the faults on the Hadoop/HBase issue tracker websites or in technical forums.

<table>
<thead>
<tr>
<th>#</th>
<th>Type</th>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Bug</td>
<td>Data Node (hdfs-1.0.4)</td>
<td>Empty Packet</td>
</tr>
<tr>
<td>2</td>
<td>Bug</td>
<td>Regionserver (hbase-90.0)</td>
<td>Distributed log splitting gets indefinitely stuck (HBASE-4007)</td>
</tr>
<tr>
<td>3</td>
<td>Misconfiguration</td>
<td>HBase Regionserver (hbase-90.0)</td>
<td>No live nodes contain current block</td>
</tr>
<tr>
<td>4</td>
<td>Misconfiguration</td>
<td>HBase Regionserver (hbase-90.0)</td>
<td>Zookeeper missed heartbeat due to paused indexed by lengthy garbage collection</td>
</tr>
</tbody>
</table>

4.5.6 False Positive Analysis

In this section, we empirically evaluate SAAD with respect to false positives. In a distributed system with complex dependencies between components, numerous external and internal factors such as network congestion, thread scheduling, I/O scheduling, etc, may inherently cause transient slowdowns or changes in execution flows. These anomalies are detected and reported by SAAD.

This inherent variability poses challenges in evaluating SAAD, because discerning anomalies caused as a result of a fault (true positives) from anomalies as a result of mis-identification (false positives) is not trivial. Moreover, some of the false positives may be due to unknown causes in the platforms studied; so the best we can do is to provide an upper bound on the potential false positive rates that SAAD signals, which we could find no known cause for.

We conduct several controlled experiments for this purpose. Each controlled experiment compares the number of anomalies detected by SAAD before and during presence of a fault under otherwise identical experimental conditions.
We evaluated SAAD extensively for 7 different fault types, one per experiment, as shown in Table 4.5 on our Cassandra cluster. For statistical significance, we repeat each experiment 10 times.

In each run, Cassandra is initialized with a baseline data set. We let the system run 30 minutes to reach stable performance. In the next 30 minutes, we let the system run without any fault injected. The anomalies detected in this period are, therefore, the result of natural variability in the system or unknown bugs. We call these anomalies false positives. We inject the fault during the third 30 minutes of the experiment. For each experiment, we measure and compare the increase in detected anomalies for the periods before and during the presence of the fault.

Table 4.5: Empirical Validation. We inject seven faults on the write path of a Cassandra node. We repeated each experiment for 10 times.

<table>
<thead>
<tr>
<th>Name</th>
<th>I/O Activity</th>
<th>Mode</th>
<th>Intensity</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>error-log-high</td>
<td>WAL</td>
<td>Error</td>
<td>High</td>
<td>Error on 100% of write operations to WAL</td>
</tr>
<tr>
<td>error-log-low</td>
<td>WAL</td>
<td>Error</td>
<td>Low</td>
<td>Error on 1% of write operations to WAL</td>
</tr>
<tr>
<td>error-memtable-high</td>
<td>MemTable</td>
<td>Error</td>
<td>High</td>
<td>Error on 100% of write operations when flushing MemTable to disk (write to SSTable)</td>
</tr>
<tr>
<td>error-memtable-low</td>
<td>MemTable</td>
<td>Error</td>
<td>Low</td>
<td>Error on 1% of write operations when flushing MemTable to disk (write to SSTable)</td>
</tr>
<tr>
<td>delay-log-high</td>
<td>WAL</td>
<td>Delay</td>
<td>High</td>
<td>Delay on 100% of write operations to WAL</td>
</tr>
<tr>
<td>delay-log-low</td>
<td>WAL</td>
<td>Delay</td>
<td>Write</td>
<td>Delay on 1% of write operations to WAL</td>
</tr>
<tr>
<td>delay-memtable-low</td>
<td>MemTable</td>
<td>Delay</td>
<td>Write</td>
<td>Delay on 1% of write operations when flushing MemTable to disk (write to SSTable)</td>
</tr>
</tbody>
</table>

Logical Anomalies. Figure 4.17(a) shows the average number of logical anomalies detected before and during the fault injection. We observe that the number of logical anomalies detected during the presence of error faults (as opposed to delay faults) is in order of magnitude higher (by a factor of 10 to 60 times) than before the fault injection. The total number of false positives of logical anomalies in all 70 runs (7 experiments each 10 runs) is only 54. In other words, the mean time between logical false positives is 38 minutes (3 false positives every two hours).

Performance Anomalies. Figure 4.17(b) shows the average number of performance anomalies detected before and during the fault injection. We observe that the number of detected performance anomalies during the presence of WAL-delay-high and MemTable-delay-low faults substantially increases (by a factor of 3 to 8 times). Anomalies do not increase during the presence of the low intensity delay fault, WAL-delay-low, since the fault affects only 1% of writes to write-ahead-logs. We observed 3 false alarms per run, or an average of a 10 minute interval between performance false positives.

4.5.7 Summary of Results

In this section, we demonstrated that SAAD

- substantially reduces the storage overhead of monitoring data
Figure 4.17: The average number of detected anomalies before and during presence of a fault (over 10 runs).

- uncovers faults that are not visible in the standard production logging level (INFO-level), with near-zero runtime overhead
- pinpoints the stages that best explain the source of a fault,
- uncovers hidden patterns in terms of task signatures instead of isolated log messages, which is effective in understanding the meaning of anomalies,
- assists users to avoid the hazard of fault-masking that can lead to major malfunction,
- proves effective in detecting real-world bugs,
- registers low false positives.

4.6 Discussion

Staged architecture. SAAD targets servers with staged architecture. Staged architecture is a well-adopted approach to build high performance servers for two reasons: i) Staged architecture leverages multi-threading support
in the operating system, as a light-weight and well-optimized mechanism to utilize multicore hardware, ii) The stage construct is a popular design pattern among developers to simplify the code structure by breaking complex logic into small and manageable building blocks. Implementing server code in easy-to-manage stages is particularly an interesting option for large code-bases that require collaboration between several independent developers. Most programming languages have built-in constructs for stages such as Executors in Java, and/or there are many third party libraries for this purpose such as Thrift [21].

**Scalability of statistical analyzer.** Our statistical analyzer is extremely light-weight. At runtime, the computation is limited to hash-map searches and statistical t-tests for each stage. We profiled the CPU utilization of the statistical analyzer while processing the stream of synopses collected from 4 Cassandra nodes with a relatively high workload (same workload as used in the experiments). On average, the cluster generated 2080 synopses per second. The CPU core running SAAD analyzer utilized approximately 0.72% of one core on an Intel Dual Core 2 processor. Based on this, we estimate that our current implementation can process synopses generated by up to 560 Cassandra nodes in real-time (29K synopses per second) on a single core. Since instances of the statistical analyzer can run independently on multiple cores, SAAD scales linearly by running an instance of the analyzer on a core, and assigning a set of servers to it for processing the synopses. In Figure 4.18 we show our estimation of cores needed to process synopses in real-time for different sizes of Cassandra cluster.

![Figure 4.18: Scalability. SAAD is light-weight and scalable.](image)

**A descriptive model to detect and diagnose elusive performance bugs, logical errors and misconfigurations.** SAAD creates a human readable, semantically augmented, hierarchical descriptive model that allows the user to easily inspect the SAAD output for hard-to-catch anomalies. The model exposes outlier tasks whether due to rare performance behaviour or execution flow in order to assist the users in their efforts to determine the source of anomalies. Since the tasks correspond to the server stages and are associated with log statements appearing in the source code, they are meaningful to the user.

**Unmasking Partial failures.** We showed that fault masking may lead to undesired circumstances. We believe that operators must be informed of failures, and leave the decision of how to deal with failures to the user. Our SAAD analyzer exposes the masked failures to the operators.

**Expensive runtime call tracing is avoided.** SAAD leverages logs statements in the code as is. In established approaches with full tracing of call graphs at run-time [49], every RPC call would be instrumented to transmit an
object which is threaded into the call chain by the original write operation on behalf of which all RPC’s execute. In contrast to full tracing of the entire application call graph, in our approach, we i) systematically instrument the source code minimally, to delimit stages to identify tasks at runtime, and ii) use existing log statements as tracepoints to track execution flow of tasks.

**Execution flow vs. data flow.** SAAD leverages log statements to track *execution flow*, and ignores the content of log messages. From log messages *data flow* can be inferred. For instance, in HDFS, all logs that report the same *block id* indicate the operations that occurred on that block, and as a result, they provide a comprehensive view of the block’s life cycle. SAAD may miss those types of anomalies that only reflect in data flows. However, data flow is highly specific to the application. It takes an expert to manually tag the data fields in the logs to form data flows. Also, tracking data requires a complex parsing mechanism to extract appropriate fields from the log records, which imposes undue overhead on the target systems.

SAAD trades the extra information that could be gained from parsing contents of log messages, with a light-weight scalable real-time anomaly detection approach.

**Limitations.** We highlight two limitations of SAAD. First, if a fault does not result in a change in the execution flows, or if the change is not visible in the logging pattern, SAAD fails to detect the anomaly associated with the fault. Second, we estimate the actual task duration as the time span between the beginning of task and the timestamp of its last visited log point. Since log points are sparsely located on the execution paths, the task may continue execution well beyond the last log point. As a result, if a fault causes a task to slowdown after the task’s last log point, SAAD is unable to detect the increase in the task duration. It must be emphasized that from our observations in the three systems we studied, in most cases, a task encounters its last log point close to its termination. For this reason, our current estimation of task duration did not become a source of inaccuracy in our analysis.

### 4.7 Conclusion

Tracing whole program execution and detailed logging imply significant run-time overheads, as well as costly post-operational log mining for anomaly detection and diagnosis. At the other extreme, anomaly detection based on production-standard logging verbosity may miss essential clues to anomalies and further complicates the root-cause analysis. In this paper, we propose a sweet-spot design which gives the best of both worlds and is directly applicable to all server codes that are based on staged architectures. We use minimal instrumentation of server code to mark stages, and track execution flows during runtime from existing log points. We introduce a light-weight real-time statistical analyzer that detects logical anomalies that manifest in the execution flow, as well as performance anomalies that affect the execution time of a system. We validated our anomaly detection technique
on three widely-used distributed storage systems, HBase, HDFS and Cassandra. We showed that our system can accurately detect logical and performance anomalies in real time.
Chapter 5

Composite Anomaly Detection using Cross-component Correlations

In this chapter, we introduce a tool that encodes semantics of structural dependencies between tiers in a multi-tier system to limit the scope of anomaly detection search.

5.1 Introduction

As system complexity grows, deploying automated tools for anomaly detection and diagnosis in large-scale multi-tier systems has become a life-saving goal for the computer industry. Many commercial tools for coordinated monitoring and control of large-scale systems exist. HP’s Openview and IBM’s Tivoli products collect and aggregate information from a variety of sources and present this information graphically to operators. However, the complexity of deployed multi-tier systems exceeds the ability of humans to diagnose and respond to problems rapidly and correctly [67].

The traditional approach to automated problem detection in multi-tier systems is to develop analytical models of system structure and behavior, which may be represented quantitatively or as a set of event-condition-action rules [106]. These models may be costly to build automatically or may require extensive knowledge about the system. If specialized domain knowledge is used, these models may either be incomplete, hence inaccurate, or may become obsolete as the system changes or encounters unprecedented situations.

In contrast, recent research [62, 57, 36, 38, 39] has investigated developing statistical models for automatic fault detection. These techniques derive probabilistic relationships, called functional invariants or correlations, between metrics captured at different points across the system. These approaches are generic and need little or no domain knowledge.
CHAPTER 5. COMPOSITE ANOMALY DETECTION USING CROSS-COMPONENT CORRELATIONS

The system raises a user-level alarm whenever it detects a significant change for one or more of these functional invariants. These approaches can be applied to a wide variety of systems, and can adapt rapidly to system changes. On the downside, they may trigger unacceptable levels of false alarms for benign changes, such as a workload mix change or an environmental setting change. Moreover, when a fault occurs, even if the fault is localized to one component, metrics collected at many other components may show abnormal functional correlations. As we will experimentally show, classifying or ranking techniques for metrics involved in correlation violations are not sufficient for pinpointing the fault location.

In this chapter, we introduce a novel system modeling technique, which combines the benefits of the above approaches, while avoiding their pitfalls.

The basis of our technique is a unifying framework for deploying heterogeneous types of models, each model tagged with semantic meaning corresponding to a particular system component or input. Each of these models uses the most appropriate approach, e.g., a generic analytical model, a statistical model or a fully specialized model, for accurately modelling the respective component or input. Our technique is flexible and allows for plug-and-play replacement of any per-component model. More importantly, we use a meta-model, called a belief network, to detect and diagnose anomalies based on the collective information gathered from all individual models. Our meta-model incorporates two key elements:

- Readily available expert knowledge about the semantic meaning and relationships between the system components, and between each component and its modeled sub-components. This information encodes the hierarchical structure of the system, including its inputs, such as environment or load, as well as their relationships.

- A Bayesian network, which makes probabilistic inferences for detecting anomalies and diagnosing the system component or input most likely to be the root cause of an observed anomaly. The Bayesian network builds on the expert knowledge of the system structure, as well as on off-line or on-line training with automated fault injection in order to form and dynamically evolve its beliefs about faulty components.

Our approach requires only high level information about the system, which typically already exists and can be easily specified. The ability to associate semantic meaning to models sensing anomalies in system components, environment or load inputs, respectively, is crucial for an accurate and meaningful anomaly diagnosis.

Expert knowledge about the meta-level relationships between system components, called prior beliefs, is desirable for enhancing the ability of the belief network to distinguish relevant events from noise and for filtering out false alarms. An example of a useful meta-level relationship is intuitively as follows: “The workload causally affects the tier(s) it feeds”. A similar meta-level correlation would be used to detect the influence of the environment, defined as configuration files, tuning knobs, and scheduled applications on various system components.
Prior beliefs can either be specified in a graphical way, by connecting components in a directed acyclic graph (DAG) through flow arcs depicting structural or input relationships, or in a high-level language, such as SML \cite{16}. The availability of prior beliefs shortens or even circumvents the need for training the Bayesian network. However, expert information does not need to be completely specified or fully accurate. Our technique enables adding new expert knowledge incrementally, and automatically prunes out or adjusts outdated beliefs about the system.

In our experimental evaluation, we use a standard dynamic Web site built using the three popular open source software packages: the Apache Web server \cite{18}, the PHP Web-scripting/application development language \cite{87}, and the MySQL database server \cite{8} running the shopping and browsing mix workloads of the TPC-W e-commerce benchmark \cite{105}, an industry-standard benchmark that models an on-line bookstore.

We define a system structure consisting of the Workload, Web server, Database server tiers, several resources in each tier, such as CPU, memory, disk, network, etc. We use a specialized model for estimating the memory footprint, based on the Miss Ratio Curve (MRC) \cite{78}. We use Gaussian Mixture Models \cite{57} for modeling other parts of our system. We train the various models using unsupervised learning and our Bayesian network using supervised learning, through automated fault injection of a variety of fault types, including disk and network hogs on system components.

Our results show that our anomaly detection method can successfully disambiguate between faults and workload or environment change cases. Our results also show that our approach can accurately pinpoint the components that are most affected by an anomaly for experiments with single or multiple faults injected into the system on-the-fly.

The rest of this chapter is organized as follows. Section 5.2 motivates our approach through preliminary experiments using generic statistical models and introduces background on Bayesian networks. Section 5.3 describes the architecture of our system and introduces our Bayesian learning technique. Section 5.4 describes our prototype and testbed, while Section 5.5 presents the results of our experiments on this platform. Section 5.6 summarizes the chapter.

## 5.2 Background and Motivation

In this section we first motivate our scheme through preliminary anomaly detection experiments with generic statistical models. We then introduce necessary background on Bayesian networks.

### 5.2.1 Motivating Example

In this section we show that a generic statistical model, which tracks correlations for metrics across the whole system cannot reliably pinpoint the faulty component. We use the common three-tier architecture of a dynamic
content Web site, consisting of a front-end Web server, an application server implementing the business logic of the site, and a back-end database. The Web site is running the shopping mix of the e-commerce TPC-W benchmark. We instrument the system to periodically sample various CPU, disk, network, and Operating System metrics for each tier. We use a model similar to Gaussian Mixture Models (GMM) \cite{57} to approximate the probabilistic distribution of pair-wise correlation between metrics, and associate a \textit{fitness score} \cite{63}, i.e., a confidence score, to each correlation.

We perform a fault injection experiment where we hog the disk on the machine running the database server. Our GMM model reports 122 violations among 36 metrics across the whole system. We then rank the metrics based on the number of correlation violations they appear in. Table \ref{table:51} shows the top 14 ranked metrics, and the number of correlation violations they appear in.

Table 5.1: Top 14 metrics involved in violations for an injected disk hog. None of the disk related metrics are among the top affected metrics.

<table>
<thead>
<tr>
<th>Rank</th>
<th>Violations</th>
<th>Metric</th>
<th>Tier</th>
<th>Component</th>
<th>Specific Metric</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20</td>
<td>DB_OS_in</td>
<td>Web</td>
<td>OS</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>18</td>
<td>DB_NET_rx_packets</td>
<td>DB</td>
<td>Net</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>16</td>
<td>WEB_NET_tx_packets</td>
<td>Web</td>
<td>Net</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>14</td>
<td>DB_NET_tx_packets</td>
<td>DB</td>
<td>Net</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>14</td>
<td>WEB_NET_rx_packets</td>
<td>Web</td>
<td>Net</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>10</td>
<td>DB_OS_cs</td>
<td>DB</td>
<td>OS</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>DB_OS_id</td>
<td>DB</td>
<td>OS</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>DB_Comp_Times</td>
<td>DB</td>
<td>OS</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>DB_IncomingQuery</td>
<td>DB</td>
<td>OS</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>8</td>
<td>WEB_OS_free</td>
<td>Web</td>
<td>OS</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>8</td>
<td>DB_IncomingRD</td>
<td>DB</td>
<td>OS</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>7</td>
<td>DB_Latency</td>
<td>DB</td>
<td>OS</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>7</td>
<td>DB_QueryThput</td>
<td>DB</td>
<td>OS</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>7</td>
<td>DB_IncomingRD</td>
<td>DB</td>
<td>OS</td>
<td></td>
</tr>
</tbody>
</table>

Each metric is named specifying the tier, the component within the tier and the particular metric captured for that component. For example, the \textit{DB_NET_tx_packets} metric represents the number of packets transmitted by the network from the database tier.

As we can see from the table, the top ranked metrics are not directly related to the most affected component, which is the database disk. The top ranked metric, the \textit{DB_OS_in} (the number of interrupts measured at the OS component on the database tier), appears in 20 violated correlations with other metrics. We further see several highly ranked metrics related to network transmit and receive packets in both the Web and Database tiers. The disk metrics do not appear in the top correlation violations mainly because of low \textit{fitness scores}.

5.2.2 Discussion of Results and Goals

The results presented show that a fault injected in one component affects system metric correlations system-wide when generic statistical models of the whole system are employed. We also show that there is no reliable method for pinpointing the faulty component when using such models.
While tagging the metrics collected with semantic meaning and ranking them based on the number of violations they are involved in may help in some cases, as was shown in previous work [57], this is not always the case. There is no simple method for associating a confidence value with a ranking result. Hence, a high level of false alarms may occur with such anomaly detection methods.

In this chapter, we address this issue by providing a method for guiding learning for the purposes of improving the accuracy of fault diagnosis in terms of both i) lowering the occurrence of false alarms and ii) accurately zooming in to the faulty component through a semantic-driven modeling approach based on Bayesian networks.

**Bayesian Networks**

A Bayesian network is a graphical model for probabilistic relationships among a set of variables [58], where dependencies among the variables are represented by a Directed Acyclic Graph. For a set of variables \( X = \{x_1, x_2, \ldots, x_N\} \), a Bayesian network encodes a set of conditional probability distributions of the form \( p(x_i \mid pa_i) \), where \( pa_i \) is the set of parents of variable \( x_i \). Given the DAG \( S \) and the set of conditional probabilities, the joint distribution for \( X \) is given by \( p(X) = \prod_{i=1}^{N} p(x_i \mid pa_i) \).

Once the Bayesian network evolves its beliefs based on prior knowledge, data, or both, we can determine various probabilities of interest from this model. The computation of a probability of interest given a model is known as probabilistic inference.

In conjunction with statistical techniques, Bayesian networks have several advantages for inference under uncertainty. Bayesian networks encode dependencies among variables, hence are robust to situations where some data entries are missing or the data sampling is noisy. The other advantage of a Bayesian network stems from the fact that it models both causal and probabilistic semantics, which makes it an ideal representation for combining prior knowledge, which often comes in causal form, and experimental data [58].

### 5.3 Semantic-Driven System Modeling Using a Belief Network

In this section we describe our approach to semantic-driven system modeling in a three tier dynamic content Web site. Our assumption is that a single component fault may occur at any given time, but with potential manifestations in other components. Our objective is to accurately pinpoint the component or system input most likely to contain the root cause of an anomaly. Towards this goal, we use a hierarchical modeling approach called a belief network combining i) heterogeneous models acting as system sensors, one per identifiable system component or input and ii) a meta-model incorporating expert knowledge into Bayesian learning.

In the following, we first introduce the overall architecture of our system and an overview of our approach. Then, we describe how the human interacts with the system by i) providing expert knowledge about system structure and the correlations between components, and ii) querying our belief network for anomaly diagnosis. Finally, we describe our meta-model based on Bayesian learning.
5.3.1 Overview

Our fault diagnosis belief network is a hierarchical network, structured as a directed acyclic graph (DAG), reflecting the underlying system architecture and the correlations between components. Figure 5.1 shows part of our layered structure belief network for dynamic content servers, which is built from the following elements: inputs, tiers, components, sensor models, and the correlations across them as follows.

The environment parameters per tier, and the workload represent inputs to the rest of the system. A node in the network represents each tier in a multi-tier system, i.e., \( T_{WL} \) - the client/workload tier, \( T_{Web} \) - the Web tier and \( T_{DB} \) - the database tier. The component layer in the network models the hardware and/or software components belonging to each tier, e.g., the disk (\( C_{DiskDb} \)), network (\( C_{NetDb} \)), etc. At the lowest level, we have a set of sensor models monitoring different system sub-components, each tagged with a clear semantic meaning, e.g., \( M_{SysDb} \) and \( M_{DbIO} \) are models sensing different aspects of the disk component of the database tier. Each sensor model is a fully fledged anomaly detection model, which expresses its confidence about detecting an anomaly in its monitored system part. The sensor models provide evidence of per-component anomalies for all upper layers of the belief network. A correlation between two nodes in the network, shown as a directed arc, encodes cause-and-effect semantics as well as structural information, and represents a dependency between the probability of anomaly of the two nodes. Correlations encode causality between components, or between a known problem and the components affected by that problem. For example, an anomaly in the database cache component will likely cause the disk component on that tier to be affected. Input correlations are also causal correlations, since an input change is likely to cause changes within the tier(s) it feeds. Furthermore, correlations encode structural dependencies, such as between a tier/component and its components/sub-components, respectively.
5.3.2 Operation

Each node in the belief network computes a confidence value i.e., probability of fault/anomaly originating within the part of the system it represents. In our example, \( T_{Web} \) and \( C_{DiskDb} \) compute probabilities of anomaly originating from within the Web tier, and from within the disk component of the database, respectively. We use Bayesian learning to infer the probability of anomaly at each node in our belief network. Bayesian networks provide a simple way to encode prior knowledge. Prior knowledge in our case is semantic structure, such as the multi-tier nature of the system, an expert’s beliefs about the relationships of interacting components, or known problems, such as, recurrent faults and the components that are affected by such faults.

The network encodes semantics and initial expert beliefs about dependencies among components is trained with data collected from the working system in experiments using fault injection. The expert knowledge about inter-node relationships makes the network converge faster during training and filters out false alarms and noise in the system. Conversely, in the process of training, initial beliefs that do not match real data are eliminated, while the belief network gains stronger confidence in valid initial beliefs. The belief network learns the statistical dependencies among components and models. It then calculates a probability of fault occurrence at each node, at any given time, based on evidence of anomalies from sensor models.

We employ a metric collection system to provide data for the various types of sensor models, including system metrics, such as network traffic statistics, OS statistics, including CPU and memory utilization, and software component statistics, such as, database statistics logs.

In the following, we describe how the system administrator interacts with our belief network by specifying expert knowledge and by querying the network for anomalies. We then describe our models and the Bayesian learning technique in this context.

5.3.3 Expert Knowledge Specification

The system administrator specifies dependencies between system components as arcs, each weighted by a dependency value. Each arc represents a correlation between the probabilities of anomaly of the two nodes involved. Each dependency value on a correlation arc indicates the correlation strength i.e., initial confidence that a correlation exists. The expert can be conservative by providing a superset of arcs. The system will prune out irrelevant correlations during training. Likewise, the initial dependency value on each arc serves only as a guidance and is adjusted based on data during training.

Depending on the underlying causal link between the nodes involved, a correlation arc may indicate that: i) a high probability of fault in one node is associated with a high probability of fault in the other node, or that ii) a high probability of fault in one node is associated with a low probability of fault in the other node. For example, if
the probability of an anomaly occurring in an input node is high, the probability that the changes perceived within
the tier it feeds indicate a fault originating within that tier is low. Likewise, if an anomaly occurs in a component,
e.g., the database cache, any anomalies perceived in the components it causally affects, e.g., the database disk,
should be ignored. In this case, the changes seen at the database disk are likely not due to a fault in the database
disk itself; hence the probability of fault in the database disk should be low. For structural correlations, a high
probability of fault occurring within a tier/component will be associated with a high probability of fault originating
from one or more of its components/sub-components, respectively.

5.3.4 Querying the Network for Anomaly Diagnosis

Querying the belief network for the purposes of anomaly diagnosis proceeds top-down through the hierarchy
embedded through dependencies in the belief network. If all input and tier nodes of the belief network show a
low probability of anomaly, then no anomaly is present, regardless of any alarms triggered by the low-level sensor
models.

If the workload tier or an environment node shows a high probability of anomaly, then no other tier would
likely show a high probability of anomaly at the same time, due to the respective input correlations. In the absence
of any workload or environment anomalies, diagnosis continues by querying the lower levels as follows. If a tier
shows a high probability of anomaly, then an anomaly is detected within that tier and per-component anomalies
of that tier are queried. A high probability of anomaly within a single tier, coupled with a high probability of fault
within a specific component of that tier pinpoints that component as containing the root cause of the anomaly.

5.3.5 Belief Network Meta-Model with Bayesian Learning

In its inferences about the per-node probability of anomaly, our belief network uses Bayesian learning of per-node
fault probabilities guided by an encoding of correlations between tiers, components and models.

The distribution over the confidence values of each model $M_i \in R$ is treated as a Gaussian distribution $N(M_i|\mu_{pa_i},\sigma_{pa_i})$, where the mean and variance are conditioned by the state of the parents of $M_i$ in the belief network.

The belief network builds on both the cumulative information of the sensor models and expert knowledge
about the relationships between components. In this section we first describe how we encode the various nodes
in the Bayesian network and the relationships between them. Then we describe our Bayesian supervised learning
scheme for evolving probabilistic beliefs about anomalies in the system based on training with fault injection.
Encoding Tiers, Components and Inputs

The state of a component or tier is represented by a two-state variable $C_i \in \{\text{anomalous}, \text{healthy}\}$ with Bernoulli distribution of the form $p(C_i = \text{anomalous}|pa_i) = \theta_{pa_i}$, and $p(C_i = \text{healthy}|pa_i) = 1 - \theta_{pa_i}$, where $pa_i$ is the state of $C_i$’s parents. The belief network can also encode known problems which are represented by Bernoulli variables $K \in \{\text{yes}, \text{no}\}$. For example, if an expert knows about a probable anomaly in processing PHP code, our belief network can encode her beliefs about which components and which models might be affected e.g., a model sensing the distribution of queries at the database tier.

Finally, our technique incorporates input variables as workload and environment variables to make the belief network aware of external changes to the system and their resulting dependencies. Input variables represent external changes such as load intensity, load-mix changes, configuration file modifications, interference due to scheduling of a third party application on a tier, etc. Changes in workload and environment are represented by Bernoulli variables $WL$ and an $ENV_{tier}$ variable per tier, respectively, which take the values $\{\text{changed}, \text{unchanged}\}$.

Encoding Dependencies between Variables

Expert knowledge is encoded as dependencies between variables and their dependency values. The structure is expressed as a directed acyclic graph (DAG). To incorporate initial expert belief about the degree of dependency of two Bernoulli variables, a prior belief with beta distribution $\beta(\alpha_{pa_i}^{\text{anomalous}}, \alpha_{pa_i}^{\text{healthy}})$ is used. The higher the $\alpha_{pa_i}^{\text{anomalous}}$ relative to $\alpha_{pa_i}^{\text{healthy}}$, the stronger the a priori belief that a component is more likely to be anomalous for a state of anomaly in its parents in the DAG, $pa_i$. Conversely, if $\alpha_{pa_i}^{\text{anomalous}}$ is substantially lower relative to $\alpha_{pa_i}^{\text{healthy}}$, then a component is unlikely to be anomalous for a given state of anomaly in its parents. We use the former type of prior belief to encode parent-child dependences between tiers and their components and between components and models. We use the latter type of prior belief to suppress false alarms due to anomalies in input. In the latter case, our belief network will automatically lower the confidence of a fault detection of a tier node if an input that feeds it, either another tier, or an external input i.e., workload or environment, shows an anomaly.

Training Using Fault Injection

Once the structure and initial degrees of dependency among components are determined by the expert, the resulting belief model is ready for training. Training has two phases: training the models, and training the belief network. To conduct the supervised training of our belief network, we perform automatic fault injection into components to observe the impact of faults on models.

Upon fault injection, the parameters of the Gaussian distribution of a variable $M_i$ evolve according to the following equations:

$$
\mu_{pa_i} = \text{average}(M_i \mid pa_i), \quad \sigma^2_{pa_i} = \text{variance}(M_i \mid pa_i)
$$

(5.1)
where $M_i \mid pa_i$ is the observed value of $M_i$ when the state of its parents is $pa_i$.

The parameters of a Bernoulli variable $C_i$ are computed as follows:

$$\theta_{pa_i} = \frac{N_{C_i=anomalous} + \alpha_{anomalous}}{N_{pa_i} + \alpha_{pa_i}}$$  \hspace{1cm} (5.2)

where $N_{pa_i} = N_{C_i=anomalous} + N_{C_i=healthy}$ and $\alpha_{pa_i} = \alpha_{anomalous} + \alpha_{healthy}$. $N_{C_i=anomalous}$ is the number of times variable $C_i$ is anomalous given the state of its parents $pa_i$, and $\alpha_{pa_i}$ is the prior belief which encodes the initial expert belief about the state of anomaly of $C_i$ given the state of its parents. The parameters of all other Bernoulli variables, i.e. $T_{tier}$, $ENV_{tier}$, and $K$ are computed in a similar manner, according to Equation 5.2.

Inferences within the Belief Network

Once the belief network is trained, it can infer the state of any component, tier, or occurrence of a known problem given the state of all the individual models. The inference about the degree of anomaly in a component, $C_i$, is made by computing the posterior probability $P(C_i = anomalous \mid M)$, where $M$ is the observation of the states of all the models. We use the Variable Elimination algorithm [46] to compute the posterior probability for variables of interest.

5.3.6 Discussion of Trade-Offs

The granularity of anomaly diagnosis directly depends on the level of detail of the system hierarchy description and the availability of models to sense each component and input of this hierarchy. For example, the belief network cannot distinguish an anomaly in the database server cache manager versus an anomaly within the application running on the database server, unless specific sensors for each are pre-defined. Hence, there is a trade-off between the burden on the system administrator for specifying expert knowledge and providing the relevant sensor models on one hand, and the usefulness of the results when querying the belief network on the other hand. However, our approach can be incrementally refined by gradually adding new models and relationships to the hierarchy. Accurate initial estimates of the degree of dependencies between nodes in the network produces fast convergence during training. However, as we will show, the network is resilient to human error and corrects any initial errors, such as extra links or inaccurate dependency values on links during training.

5.4 Prototype and Testbed

We implement the Bayesian inference engine and all other models in Java. We use two types of models as our system sensors: a specialized resource model based on the Miss Ratio Curve (MRC) for sensing memory footprint
anomalies, and Gaussian Mixture Models (GMM) for all other system components. We describe the MRC and GMM models next.

MRC Model: The miss-ratio curve (MRC) of an application shows the page miss-ratios at various amounts of physical memory. This approach was first used in cache simulation and was recently proposed for dynamic memory management [115]. The MRC reveals information about an application’s memory demand, and can be used to predict the page miss ratio for an application, given a particular amount of memory. The MRC can be computed dynamically through Mattson’s Stack Algorithm [78]. The algorithm is based on the inclusion property, which states that a memory of k + 1 pages includes the contents of a memory of k pages. The popular Least Recently Used (LRU) algorithm exhibits the inclusion property, thus allowing us to estimate the miss ratio for a given amount of memory m.

We employ the MRC to model the memory footprint of the database server. We sample the MRC at various points in time and we compute the vector of mean ($\mu_{\text{mrc}(i)}$) and variance ($\sigma_{\text{mrc}(i)}^2$) of the sampled curves for $0 \leq i \leq e$, where e is the effective size of memory. We use the following equation to measure the proximity of the observed MRC to the expected/normal curve:

$$\text{dist} = \frac{1}{e} \sum_{i=0}^{e} \frac{(\text{MRC}(i) - \mu_{\text{mrc}(i)})^2}{\sigma_{\text{mrc}(i)}^2}$$ (5.3)

Gaussian Mixture Models: Gaussian Mixture Models (GMM), a clustering algorithm, is used to capture the statistical correlation between pairs of metrics [36, 62].

In contrast to previous work on GMM [57] which determines statistical correlations between all pairs of metrics, we break down metrics into subsets, each subset corresponding to a specified system part. Each model represents a semantically meaningful system part; it derives pairwise correlations among the subset of metrics belonging to its respective system part, as well as between this subset of metrics and the rest of other metrics. This enables us to model behavior of components of interest. For example, to model the disk I/O, we derive the correlations between metrics pertaining to disk as well as correlations between disk metrics and all other metrics. Collectively these correlations form the basis for the disk I/O model, which raises alarms with a certain confidence if any of these correlations are violated. In our experiments, we use GMM to model Disk I/O and Network.

Training the Models: Training of each model can be either supervised or unsupervised, depending on the type of the model. In this work, we train each model individually, in an unsupervised manner. The size of the data needed for each model is also different. For example, the MRC model typically uses a trace of 1 million page accesses, which might take a couple of minutes to collect, while GMM-based models need a set of samples collected over a 30 minute period, on average, in order to achieve a stable model.
Testbed and Statistics Collection: We use the industry-standard TPC-W e-commerce benchmark running the browsing and shopping workload mixes. TPC-W is implemented using three popular open source software packages: the Apache Web server, the PHP Web-scripting/application development language and the MySQL database server. We use the Apache 1.3.31 Web server in conjunction with the PHP module. We use MySQL 4.0 with InnoDB tables as the database back end. We use a client emulator which drives our experiments. Our experimental testbed is a cluster of Xeon Intel computer with 2GB of RAM and four 3.00GHz CPUs. All the machines use the Ubuntu Linux operating system. All nodes are connected through 1Gbps Ethernet LAN.

We use lightweight tools for measuring system-level and application-level metrics. Monitored metrics include OS, network, database, and workload statistics. The data is obtained from `vmstat`, `netstat`, instrumented `Mysql`, and the workload emulator. We use a scheduler interposed between Apache-PHP and Mysql to collect metrics of interest about queries and database statistics, such as throughput and active connections. Logs obtained from the client emulator are used to model workload characteristics. We employ a central data collection layer which collects metrics from various points in the system online, and after preprocessing provides it to the models. To facilitate data collections daemons residing on each tier and inside applications of interest collect data and send it to the centralized data collection layer.

5.5 Experiments

In this section, we validate our anomaly diagnosis approach through several experiments. We first conduct experiments with a single system-level fault or workload mix change, then with multiple (two) simultaneous system-level faults showing the diagnosis capabilities of our system. Finally, we induce an application-level change unknown to the system, and show how our system detects the most affected component.

The structure of the belief network used in these experiments is shown in Figure 5.1. We consider five components in our experimental belief network: `C_{WebDisk}`, `C_{WebNet}`, `C_{DbDisk}`, `C_{DbNet}` and `C_{DbCache}` i.e., the disk, and network on the Web tier, and the disk, network and database cache, on the database tier, respectively. We use 8 models. Except for \(M_{DbMrc}\), modeling the database cache, all models are based on GMM.

Single Fault Diagnosis. We train and evaluate the belief network with injecting two types of faults: network hogs and disk hogs. In addition, we induce load mix changes. Network hogs are induced by flooding the network with useless packets to a victim machine. A disk hog is implemented by spawning a thread running the Unix command `dd` on a victim machine, which reads a large file and copies it to the `/tmp` directory. We induce disk and network hogs on the Web and database machines, in separate experiments. We also induce a load mix change, from browsing to shopping, while only the browsing mix is used during training. Figure 5.2 shows the reaction of all 8 low-level sensor models to each type of fault. Table 5.2 shows the corresponding results of fault
diagnosis at the upper levels in the belief network for each type of fault, respectively. Each row in the table corresponds to a fault type. Each element in the row indicates the confidence/probability of fault in either a tier or component corresponding to each column of the table. This data shows that our scheme can diagnose the anomalous component with high degree of confidence in each case of anomaly. The table also shows that, when there is no anomaly in the system, all tiers indicate a low probability of fault.

The results of the experiment for a load mix change show the ability of our scheme to avoid false alarms in cases of an unencountered, but benign change in workload. As we can see from Figure 5.2 (LoadMix), out of the 8 models, the M_LoadMix and M_DbSys models show the highest probability of anomaly. Correspondingly, in Table 5.2 (Load Mix), C_DbDisk (database disk) shows a high probability of anomaly. However, since the dependencies between tiers have been encoded in the network, the database tier shows a relatively low (40%) confidence in detecting an anomaly originating within that tier. Hence, our scheme diagnoses the change in the workload, as detected by T_WL, as the reason for the change in behavior seen at the database disk.

![Model responses to various faults.](image)

**Multiple Simultaneous Faults.** We inject two of the disk and network hogs simultaneously, on-line, while the network is trained using single fault injection as before. The diagnosis is difficult because the faults manifest...
in different sensor models. Table 5.3 shows the fault diagnosis of our network for all cases. As expected, the confidence at the tier-level is very high when both faults are injected in the same tier. In the case of fault injection into different tiers, the dependency between tiers results in a 56% confidence at the tier level for both tiers i.e., the belief network cannot determine the tier where the fault originates from, as expected, but the belief network has high confidence at the component level.

**Unknown Fault.** In this experiment, we show the ability of our model to detect a type of fault not seen during training, i.e., an induced anomaly in the memory footprint of the database cache (modeled using the MRC model). We substantially change the pattern of the frequent Bestseller query in the TPC-W benchmark. The new query pattern increases the memory footprint of the frequent query, which however still fits in the database cache. In this experiment, we disable models directly sensitive to the root cause of this fault. This includes disabling all models sensing environment changes and query access pattern changes within the workload, which would otherwise pinpoint the top level cause of anomaly to an input/configuration change as in the load mix change experiment.

Due to the frequency of the query, the anomaly manifests itself in several components of both tiers. For example, the database cache is affected, but the network on the Web tier is affected as well, due to an increase in the response size of this query. Almost all model responses record a normalized deviation between 30% and 150% from the average NoFault situation, which makes diagnosis of the faulty component non-trivial.

Table 5.4: **Problem diagnosed in the DB tier within the DB cache.**

<table>
<thead>
<tr>
<th>Component</th>
<th>Confidence</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_{tb}$</td>
<td>0.93</td>
</tr>
<tr>
<td>$C_{dbCache}$</td>
<td>1.00</td>
</tr>
<tr>
<td>$C_{dbNet}$</td>
<td>0.61</td>
</tr>
<tr>
<td>$C_{dbDisk}$</td>
<td>0.00</td>
</tr>
<tr>
<td>$T_{Web}$</td>
<td>0.64</td>
</tr>
<tr>
<td>$C_{WebNet}$</td>
<td>1.00</td>
</tr>
<tr>
<td>$C_{WebDisk}$</td>
<td>0.00</td>
</tr>
</tbody>
</table>

Overall, as we can see from Table 5.4 our belief network makes a diagnosis of fault within the database tier, rather than the Web tier in this case. The intuition behind the higher fault probability at the database tier is that two of the components at the database tier show an anomaly. Moreover, the memory footprint of the Bestseller query is very stable during training. In contrast, the recorded statistical variation of all other models is substantially
higher. Since inferences are made based on training data, as well as evidence of fault from all sensor models, our belief network identifies the fault as occurring within the database tier, specifically within the database cache ($C_{DbCache} = 1.0$). This pinpoints the component most affected by the root cause of the anomaly, hence providing valuable information in the absence of the root cause itself, which the system cannot detect in this case.

**Pruning Irrelevant Dependencies.** In this experiment, we show that our network is resilient to mistakes of the human expert or any irrelevant initial links. We define an (invalid) dependency between the $C_{DbNet}$ component and the database I/O model ($M_{DbIO}$) and re-train the belief network. After re-training, Figure 5.3 shows that the network has automatically modified the dependency value between $C_{DbNet}$ and $M_{DbIO}$. To show this, we plot the inferred log probability of anomaly in the disk and network, $C_{DbDisk}$ and $C_{DbNet}$, while decreasing the probability of fault (confidence) as sensed by the database I/O model from 1 to 0. We can see that the inferred probability of fault in the DB network ($C_{DbNet}$) is very small (almost zero, due to the log scale), regardless of the probability of fault detection read from $M_{DbIO}$. This is very different from a component with a true dependency to $M_{DbIO}$.

We show the variation of the probability in $C_{DbDisk}$ as measured during the same experiment, which decreases significantly with the decrease in probability in $M_{DbIO}$. This shows that after training, a wrong link is pruned by automatically adjusting the degree of dependency to a value which no longer influences the inference at $C_{DbNet}$. This reduces the burden of the expert by allowing for placing extra (default) links, or random dependency values on links in the initial network.

### 5.6 Summary

In this chapter, we introduce a novel semantic-driven approach to anomaly detection and diagnosis. Our approach bridges and integrates heterogeneous per-component models through a Bayesian network encoding the semantic meaning of components, their interactions and the association between components and models.
The Bayesian network is trained by observing the reaction of each model to injected faults. Both per-
component sensor models and the Bayesian meta-model thus evolve dynamically as the system or application 
changes. The close correspondence between the structure of the meta-model and the structure of the system facil-
itates diagnosis of the component that most likely contains the root cause of a perceived anomaly. This allows us 
to differentiate component faults from cases of external changes of workload or environment.

We perform experiments using the industry-standard TPC-W benchmark and a wide range of fault injection 
scenarios, including disk, network and application anomalies, as well as single and multiple faults. We show 
that our approach provides accurate diagnosis, while requiring minimal human expert intervention. Providing 
readily available information on system structure and dependencies results in rapid convergence during training. 
Conversely, the network is resilient to human error and prunes out or adjusts incorrect dependency information.
Chapter 6

Related Work

6.1 Black-box Statistical Approaches

The black-box statistical approaches assume that the system is mostly correct and define anomalies as statistical deviations from the norm.

Jiang et al. [62] and Chen et al. [36] introduce an automatic fault diagnosis scheme based on pairwise statistical correlation of monitoring data. They define the concept of invariant as a correlation between a pair of metrics which holds under various load intensities. Invariants are used as the basis of anomaly detection. Unlike our scheme, their work does not incorporate domain knowledge in the process of discovering of the invariants.

Aguilera et al. [23] introduce a statistical method for fault detection in multi-tier Internet services. Normal behaviour is inferred through monitoring packet exchange between tiers. This method suffers from high false positives and low accuracy, needs substantial volume of monitoring data for training, and it is highly sensitive to slight changes in execution flow and interaction topology of components.

Some blackbox approaches to anomaly detection are based on peer comparison [85, 66]. PeerPressure [107] is an automatic misconfiguration troubleshooting tool using Bayesian inference and statistical reasoning based on peer configuration settings. This method is based on the idea that peers have similar behaviours. By tracking configurations of a set of similar Windows workstations, PeerPressure builds a statistical model to predict correctness of Windows registry entries.

Cohen et al. [40] correlate system metrics to high-level states, and leverage the model to find the root-cause of faults based on past failure cases within similar settings [41]. The key idea is to deploy a graphical statistical model, Tree Augmented Bayesian Networks (TAN), to characterize high level system state in terms of
relationships between low level metrics. The TAN graphical model provides insights into the underlying cause of faults and provides a unified abstraction to encode past cases of failures for future mapping between symptoms and root-causes. Similarly, Lao et al. [111] identify known problems by mapping traces of function calls to known high level problems and collect a knowledge-base of association between symptoms and root-causes.

Liblit et al. [73] sample traces from multiple instances of a program from multiple users to limit the overhead of monitoring on each user. They use classification techniques to predict correlation between the traces and known bugs.

In contrast to methods introduced in Chapter 3 and Chapter 5, these methods study correlations and statistical deviations agnostic to semantics of the system.

6.2 Theoretical Model-Based Approaches

Theoretical model-based approaches [92, 102, 27, 99] leverage theoretical concepts, such as Queuing theory [56, 61] to contrast observed system-behavior with theoretically expected behaviour. There are three major drawbacks to model-based approaches. First, they are highly sensitive to the parameters under which the system operates, and these parameters often change frequently. As a result, models are brittle and prone to become obsolete. Second, they require a detailed and often laborious process of creating the models. The complex modelling makes them prone to error or in some cases oversimplifying by overlooking corner cases. Third, theoretical modeling makes certain assumptions about the behaviour of workload and/or components in order to simplify mathematical analysis. For instance, most models based on Queuing theory assume independence between job arrivals.

To mitigate brittleness of theoretical modeling, Threska et al. [102] introduce Ironmodel, which is an adaptive model that adjusts its parameters based on the context in which the system operates in. However, it fails to mitigate the modelling complexity and the unrealistic assumptions of system behaviour.

With SelfTalk and Dena (Chapter 3), users do not need to go through a complex modelling process. While hypotheses require an understanding of the system, we do not require the relationships described by the hypotheses to be always correct, and can inform the user of their validity.

6.3 Expectation-Based Approaches

System programmers, as well as system analysts, can detect anomalies through expressing their expectations about the system’s communication structure, timing, and resource consumption. Language based approaches include MACE [68], TLA+ [70], PCL [79], PSpec [86] and Pip [88]. PSpec [86] is a performance checking assertion language that allows system designers to verify their expectations about a wide range of performance
properties. The type of assertions of PSpec is similar to SelfTalk comparison relations. Also, similar to SelfTalk, PSpec uses a relational approach to represent and query monitoring data [94]. However, PSpec lacks the ability to use mathematical functions as the basis of checking the behavior of the system. Similar to our work, Pip [88] is an infrastructure for comparing actual behavior and expected behavior of a distributed system, expressed through a declarative language. However, unlike our work, Pip requires the source code to be significantly modified by adding some special annotations. In general, in contrast to the existing language based approaches, our work mainly targets users such as performance analysts who have a general insight into the system’s behavior but lack the knowledge of the details and have no access to the system’s source code.

Performance analysis tools [44, 59, 12, 48, 76, 17] allow programmers to analyze the performance of a system to find sources of inefficiencies. These approaches are useful for a programmer who has detailed insight about the system and has access to fine-grained profiling. While SelfTalk/Dena can be used for the same purposes, they also provide, through hypotheses, a common knowledge shared by the parties involved in different life cycles of a system, mainly post-deployment where effective maintenance is the priority.

Chopstix [28] is a rule-based anomaly detection system that collects "sketches" to efficiently gather long term monitoring data with low-overhead. The anomalies are detected based on a set of rules defined by operators. SelfTalk and Dena are complimentary to Chopstix by providing flexibility to generate the rules.

6.4 Mining Operational Logs

DISTALYZER [80] provides a tool for developers to compare a set of baseline logs with normal performance to another set with anomalous performance. It categorizes logs into event logs and state logs. Similar to our approach, it uses timing and frequency of event logs to create features that capture the performance of the system. In addition, it includes the values reported in the state logs as part of the diagnosis process. This approach can be used for a variety of systems, since it is agnostic to the system internals. However, it relies on ad-hoc methods, such as thread ids, to group log messages. SAAD leverages the structure of components and common thread communication patterns, to automatically determine the association between log records and tasks, with high precision. Moreover, our approach targets production systems in online mode by performing log analysis in real time.

Xu et al. [109] leverage the schema of logging statements in the source code to parse log records. Since the log template of each log record can be determined, a set of rich features, including the frequency of logs with the same type, can be extracted from the log sets. They apply principal component analysis (PCA) to detect anomalous patterns. However, in their work, grouping log messages is performed using ad-hoc criteria, and requires regular expression text processing to associate log messages to the log points in the code. Regular expression matching
is extremely expensive, and not suitable for online analysis. Also, the outcome of PCA analysis does not produce a semantically rich result. To mitigate this aspect, they present a brief description of anomalies in a decision tree. But it is not always possible to generate a meaningful decision tree. In contrast, SAAD is completely white-box, because it leverages the internal structure of components. In our model, the concept of stages provides semantic richness to the analysis. In addition, unlike PCA analysis used by Xu et al. [109], which suffers from the curse of dimensionality, our statistical model is lightweight, and can be constructed and used on-the-fly.

Fu et al. [51] use Finite State Automata to learn the flow of a normal execution from a baseline log set, and use it to detect anomalies in performance in a new log set. Like our approach, they use timing of log records to build a performance model. Similar to other off-line approaches [110] [74] [75] [98], they use text mining techniques to convert unstructured text messages in log files to log keys, which requires expensive text processing. Moreover, the accuracy of their results is highly sensitive to the heuristics used for text processing. Tan et al. [101] also adopt a FSA approach to model the performance of Hadoop MapReduce clusters from the logs. Filtering-based approaches [81] [72] map the sequences of log messages to system administrator alerts in case of incidents.

With LogEnhancer, Yuan et al. [114] introduce techniques to augment logging information to enhance diagnosability.

### 6.5 Inferring Execution-Flow

Sambasivan et al. [90], Pip [88], and PinPoint [38] diagnose anomalies by comparing the expected execution flow with the anomalous one. The common denominator of these approaches is detailed fine-grained tracing, which is provided by explicit code instrumentation, or thorough distributed tracing enablers, such as, x-trace [49], Dapper [93] and Magpie [27]. Due to high monitoring resolution and causality information, trace-based anomaly detection techniques are more precise than regular log-based approaches. However, tracing generates a large amount of data, which needs to be processed offline. Also, because of the high resolution data collection, they impose undue runtime overhead. As a result, in a production system, tracing is used in a limited way, only for short intervals of debugging.

Sherlog [112] infers likely paths of execution from log messages to assist the operator in diagnosing the root-cause of failures. Unlike SAAD, which is designed for real-time anomaly detection, SherLog is a postmortem error diagnosis tool. SAAD is orthogonal to Sherlog. Users may use SAAD to identify anomalous execution paths, as an input to Sherlog to analyze the control and data execution flow leading to an error.

**Trace Analysis.** Trace analysis analyzes runtime execution traces by instrumented programs to detect anomalies or diagnose faults. These methods require modifying the server statically by adding instrumentation to the code, or dynamically by modifying the binary image at runtime. Such modifications incur runtime overhead and may
not always be possible in production systems. ConfAid\cite{25} and X-ray\cite{24}, for example, causally trace the failure symptom to the culprit misconfiguration entry. ConfAid applies backward taint analysis to track an erroneous message to the most likely configuration parameter. This approach needs to run on a dynamic instrumentation framework such as Intel’s Pin\cite{13} to collect high-resolution execution traces at the instruction level. In contrast, SAAD leverages the existing log statements in the code as a means of tracking execution flows. The trace analysis methods assume that the failure is reproducible in order to conduct the causal tracing. In a distributed system, such as HBase, however, faults are hardly reproducible, and recreating the circumstances leading to a failure is nontrivial.

6.6 Inferring Causality

Sherlock\cite{26} and its successor NetMedic\cite{63} infer the source of a problem in an enterprise network based on causal dependency. Aguilera et al.\cite{23} find a culprit node which is responsible for high delay, by inferring the dominant causal paths in a multi-tiered system. Onilner et al.\cite{82} propose a method for identifying the sources of problems in complex production systems, where, due to the prohibitive costs of instrumentation, the data available for analysis may be noisy or incomplete. This approach infers the influences among components in a system by looking for pairs of components with time-correlated anomalous behavior. The results are summarized by the strength and directionality of shared influences through a Structure-of-Influence Graph (SIG). The accuracy of causality-based approaches is highly sensitive to the quality of monitoring data to construct the trace of individual requests spanning multiple tiers.
Chapter 7

Conclusion and Future Work

In this chapter we sum up the key ideas and techniques presented in this dissertation. Then, we present our vision on how these techniques can be extended as future work.

7.1 Summary

Online distributed systems have become the backbone of many modern business operations. As these systems are designed to become scalable and grow in complexity, it is essential to study mechanisms that detect and prevent failures in these systems.

Although current systems tolerate failures and provide an illusion of reliability while running on top of otherwise unreliable commodity hardware, users still need tools to investigate the cause of failures. If a failure goes undetected, it can potentially inflict serious damage to business operations in the form of a general outage or data loss.

In this dissertation, we introduced a set of tools that leverage the semantics of a system in order to detect unusual patterns and present them to the user in a human-understandable way. The semantics of the system are expressed through a set of high level beliefs or invariants provided by the user in an expressive language, or through minimal instrumentation that exposes the known structure of the system. Specifically, we introduced, prototyped and evaluated three systems that demonstrate the functional benefits of our overall thesis in practical scenarios.

First, we introduced SelfTalk and Dena, a query language and a runtime environment that allow users to express their hypotheses about the behavior of a system in a high-level descriptive language, and verify it with monitoring data. We evaluated SelfTalk and Dena on a multi-tier storage system typically used in enterprise systems.
Second, we introduced Stage-aware Anomaly Detection (SAAD) that analyzes execution flows based on tracking operational log points in real time. SAAD takes advantage of the staged architecture of servers which break down the operations presented to them into tasks and execute them in stages. SAAD infers execution flow and duration of tasks from tracking operational log points on-the-fly, with minimal overhead on a running system. We evaluated SAAD on three distributed storage systems: Hadoop distributed file system (HDFS), HBase, and Cassandra.

Finally, we proposed a framework that allows users to combine multiple anomaly detection methods into a unified belief network that encodes the structural dependency between system components. We demonstrated that our approach provides accurate diagnosis of problems in a multi-tier server system typically used in enterprise settings.

7.2 Future Work

The integral vision of this dissertation is to leverage semantic knowledge of the inner workings of a system for diagnosing failures. Based on this vision, in the following, we present future extensions of each of the three techniques presented in our work.

7.2.1 SelfTalk and Dena

We have developed SelfTalk and Dena, a descriptive language and a run-time system, which work in conjunction to enable users to diagnose and validate the performance and behavior of multi-tier storage systems; our tools allow administrators to express their belief in the form of descriptive queries, and verify them with monitoring information.

An immediate extension to this work is to build a knowledge-base of hypotheses which can be shared between different deployments of the same infrastructure. We believe that separation of hypothesis and expectation (descriptive models and the instances of the model based on monitoring data) is the key enabler for porting hypotheses between different infrastructures.

Another extension to this work is to enable defining relationships between hypotheses. This is a natural way of understanding the behaviour of a system as a cause-effect network. For instance, a Miss Ratio Curve hypothesis is closely related to the performance of a disk. A high miss rate increases the chances that blocks issued to disk are sequentially adjacent, which effectively reduces the latency of disk access, although the overall performance suffers. The parameters of the disk model must be adjusted accordingly to cover cases of sequential blocks as a result of a high miss ratio.
SelfTalk is a declarative language with a syntax similar to SQL. We believe that SQL-like languages are easier for system administrators to understand, because of their familiarity with SQL. However, declarative languages like SQL are not suitable for iterative algorithms. We hide the complexity of the relation function by allowing the user to define the relations as pre-made functions in an imperative language like Matlab or Ruby. We believe that a useful extension to this work is to allow users to use a hybrid of object and functional languages to express the hypothesis. Languages like OCaml [10], Scala [14], LINQ [7] have proved effective to express complex concepts in concise, easy-to-understand syntax.

7.2.2 SAAD Analyzer

Stage-Aware Anomaly Detection (SAAD) analyzes operational log patterns in real-time and uncovers hidden anomalies based on the execution flow of tasks. SAAD is capable of detecting semantically rich patterns without mining the large volume of log data itself, thus enabling real-time detection of unusual logging patterns.

An immediate extension to this work is to extend the analysis to data flow, for example block_id in HDFS logs. Unlike execution flow, which can be tracked directly from the life-cycle of running threads and inter-thread communication, tracking data flow requires accessing and interpreting fields of data in log records. We believe that log statements in the source code need to be modified to generate structured records with annotations, to circumvent text processing during runtime. Clustering and anomaly detection must extend the feature set beyond using log frequency vectors and durations to include extracted data from execution of tasks.

Another possible extension to our SAAD analyzer is to infer causality from temporal correlations between outlier tasks. The root cause of a failure can be better understood by tracking the causality between anomalous patterns. For instance, a failure of a HDFS Data Node in accessing a block may affect a HBase region server hosted on a different node. The anomaly appears on both hosts. Leveraging SAAD, we could infer the causality by tracking the start and end of outlier tasks on each node [23].

Our SAAD logger module can be extended to collect the delay caused by queuing. Currently, we consider the duration of tasks as the metric of performance slow-down. However, a slow-down can occur due to increase of wait-time in the inter-stage queues. Including the wait time in the pattern recording of the SAAD analyzer would improve the precision of detecting performance outliers.

7.2.3 Composite Anomaly Detector using Cross-Component Correlations

Our composite anomaly detector leverages the relationships between the behavior of various components to instantly verify the potential factors that cause the system to malfunction. Exposing relationships between components explicitly to anomaly detection is important in investigating the root cause of a fault. Run-time monitoring of system metrics alone cannot guide users to diagnose a faulty component, because a fault in one component
may register as unusual patterns for many metrics throughout the system. We envision that a closer integration of
the two new tools we introduced in this dissertation (SelfTalk/Dena and the SAAD analyzer) is necessary in order
to provide the user a well-ordered end-to-end process for root-cause analysis of otherwise hard to diagnose faults.
Bibliography


