Predicting Failures in Complex Multi-Tier Systems

Doctoral Dissertation submitted to the
Faculty of Informatics of the *Università della Svizzera italiana*
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

presented by
Rui Xin

under the supervision of
Prof. Mauro Pezzè

November 2020

Dissertation Committee

Prof. Walter Binder	USI Università della Svizzera italiana, Switzerland
Prof. Cesare Pautasso	USI Università della Svizzera italiana, Switzerland
Prof. Holger Giese	University of Potsdam
Prof. Rogério de Lemos	University of Kent

Dissertation accepted on 18 November 2020

Prof. Mauro Pezzè
Research Advisor
USI Università della Svizzera italiana, Switzerland

Prof. Walter Binder
PhD Program Director

Prof. Silvia Santini
PhD Program Director

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Rui Xin
Lugano, 18 November 2020

Abstract

Complex multi-tier systems are composed of many distributed machines, feature multi-layer architecture and offer different types of services. Shared complex multi-tier systems, such as cloud systems, reduce costs and improves resource utilization efficiency, with a considerable amount of complexity and dynamics that challenge the reliability of the system.

The new challenges of complex multi-tier systems motivate a new holistic self-healing approach, which must be accurate, lightweight and proactive, to ensure reliable cloud applications. Self-healing techniques work at runtime, thus they offer automatic and flexible ways to increase reliability by detecting errors, diagnosing errors, and either fixing the errors or mitigating their effects. Self-Healing Systems leverage the time between the activation of a fault and the failure by taking actions to avoid failures. Self-Healing systems shall predict failures, localize the faults and fix or mask them before the failure occurrence.

In my Ph.D, I focused on predicting failures and localizing faults. In this thesis I present an approach, *DyFAULT*, that predicts failures by detecting anomalous systems states early enough to diagnose the causing errors and fix them before the failure occurrence, and localizes faults by leveraging the collected data to pinpoint the location of error and possibly the type of the fault.

The contribution of my Ph.D work includes: (i) an approach to accurately predict failures and localize faults that requires training with fault seeding. (ii) an approach to predict failures and localize faults that requires training with data from normal execution only. (iii) a prototype implementation of the two approaches (iv) a set of experimental results that evaluate the proposed approaches of *DyFAULT*.

Acknowledgements

I would like to express my thanks to my advisor, Prof. Mauro Pezzè, for his thorough support in guiding the project and improving my academic skills. Without his considerate help in both work and life, I would not have been able to complete this research.

I also would like to thank a few researchers from Università della Svizzera italiana, including Prof. Antonio Carzaniga, Prof. Walter Binder, and my colleagues in the STAR research group, for many talks that extended my knowledge and inspired my philosophical thoughts.

Thanks also to Yudi Zheng, Haiyang Sun and Jingjing Lin, who helped me out when I had personal issues in the midst of my thesis writing.

The four and a half years' study at Università della Svizzera italiana has offered me a peaceful time window in my life, that gave me many retrospective moments to gain both the motivation to keep my curiosity about the world, and the rationality to eliminate the fear when facing unknown. I deeply appreciate University of Lugano.

Contents

Figures

Tables

Chapter 1

Introduction

Modern computing power boosts in different dimensions: Horizontally, hardware configuration improves as industrial growth benefiting from new techniques; Vertically, administrators distribute computing tasks, queries, storage and other resources to multiple machines.

A popular solution of organizing and managing geographically distributed machines is through multi-tier systems that represent the backbone of many contemporary systems, notably cloud systems. Cloud multi-tier systems, hereafter *cloud systems*, enable ubiquitous, convenient and on-demand network access to a shared pool of configurable computing resources, for instance, networks, servers, storage, applications and services. They provide on-demand self-service, broad network access, resource pooling, rapid elasticity, and measured service [84]. Cloud multi-tier systems are usually composed of various types of physically distributed machines, and feature multi-layer architecture, typically "Server-Infrastructure-Platform-Application" from bottom up. Cloud systems offer different types of service, namely Software as a Service (SaaS), Platform as a Service (PaaS) and Infrastructure as a Service (IaaS) [64]..

Distributed multi-tier architectures bring considerable amount of complexity and dynamics to the system, hence challenging the reliability of the system [7]. Risks introduced by new characteristics of distributed multi-tier architectures include: (i) On-demand service enables the provision of services in a flexible way, but imposes new reliability requirements on the service provision to keep the cloud running smoothly; (ii) Distribution of resources over wide area networks results in new network reliability and latency requirements; (iii) Resource and service virtualization provides resource pooling that abstracts the hardware layers, thus increasing the criticality of the reliability of virtualization; (iv) Computing elasticity enables rapid dynamic configuration changes by runtime adjustment of both resources and services, but introduces new reliability requirements, such as latency of the adjustments and elasticity failures.

Classic pre-deployment testing [121] and analysis [104] techniques do not fully address the many challenges of cloud systems. As Buyya et al. point out: *cloud management in an autonomic manner is necessary for reliable cloud service, and self-healing is an important aspect of autonomic management* [11]: The new challenges of complex distributed multi-tier systems motivate a new holistic self-healing approach, which must be accurate, lightweight and proactive, to ensure reliable cloud applications. Self-healing techniques work at runtime, thus they offer automatic and flexible ways to increase reliability by detecting errors, diagnosing errors, and either fixing the errors or mitigating their effects [60].

Following the terminologies introduced by Avizienis et al. [4] that we summarize in Table 1.1,

we present self-healing workflow on the timeline of fault and failure. The activation of a fault, such as a memory leak, corrupts part of the total states, and may later lead to a failure.

As Figure 1.1 shows, a typical **Self-Healing System** leverage the time between fault activation and expected failure by taking actions to avoid failures. With respect to the stages identified by Huebscher et al. [60], self-healing approaches are composed of three phases: (i) the **failure prediction** phase checks whether the system has erroneous states and whether such states may cause some failures, and indicates potential failures, by issuing *failure alerts* that triggers the fault localization phase. (ii) the **fault localization** phase diagnoses the errors, localizes the faults, and triggers the error fixing phase. (iii) The **Error Fixing** recover the system from erroneous states to normal states and make the running system deliver correct service.

We define the time interval between a fault activation and the failure prediction as *prediction time* ($T_{prediction}$), the time interval between a failure prediction and a fault localization as *localization time* ($T_{localization}$), and the time between *Fault Localization* and the time the *Failure* would occur with no fixing as *reflex time* (T_{reflex}).

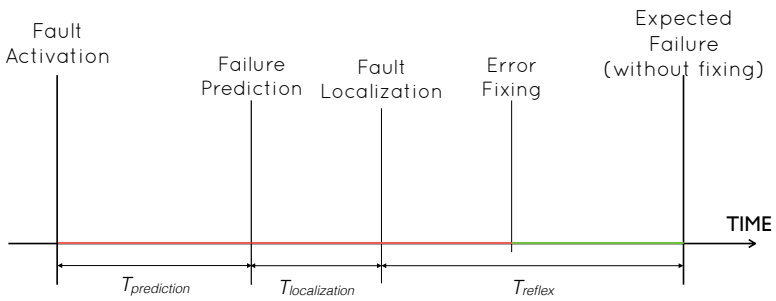

Figure 1.1. Fault and failure timeline

In this dissertation, I presents the results of investigating solutions for *predicting failures* and *localizing faults*. My work targets the IaaS level and my research work is grounded on (i) the observation that self-healing approaches can address well failures that are unavoidable in the context of multi-tier applications, and emerge due to the characteristics on multi-tier environments that can be only partially reproduced in testing environments, and (ii) the limitations of current self-healing approaches that build upon assumptions not always valid in the multi-tier domain.

In my PhD research, I studied and implemented an approach, *DyFAULT*, that addresses the new characteristics of cloud systems by integrating techniques for predicting failures and localizing faults. The new holistic approach: (i) *predicts failures* by detecting anomalous systems

System	A system is an entity that interacts with other entities, i.e., other systems, including hardware, software, humans, and the physical world with its natural phenomena.
Envrionment	The environment refers to the other systems among the entities that the given system interacts with.
Function	The function of a system is what the system is intended to do and is described by the functional specification in terms of functionality and performance.
Behavior	The behavior of a system is what the system does to implement its function and is described by a sequence of states.
Total state	The total state of a given system is the set of the following states: computation, communication, stored information, interconnection, and physical condition.
Structure	The structure of a system is what enables it to generate the behavior.
Component	From a structural viewpoint, a system is composed of a set of components bound together in order to interact, where each component is another system, etc.
Service	The service delivered by a system (in its role as a provider) is its behavior as it is perceived by its user(s).
User	A user is another system that receives service from the provider.
System Boundary	The part of the provider's system boundary where service delivery takes place is the provider's service interface.
External State and Internal State	The part of the provider's total state that is perceivable at the service interface is its external state; the remaining part is its internal state.
Use Interface	The interface of the user at which the user receives service is the use interface.
Correct Service	Correct service is delivered when the service implements the system function.
Failure	A service failure, often abbreviated here to failure, is an event that occurs when the delivered service deviates from correct service.
Error	An error is the part of the total state of the system that may lead to its subsequent service failure.
Fault	The adjudged or hypothesized cause of an error is called a fault.
Error Detection	An error is detected if its presence is indicated by an error message or error signal

Table 1.1. Basic Terminology

states early enough to diagnose the causing errors and fix them before the failure occurrence, (ii) *localizes faults* by leveraging the collected data to pinpoint the location of error and possibly the type of the fault.

The contribution of my Ph.D work includes:

- an approach to accurately predict failures and localize faults that requires training with fault seeding.
- an approach to predict failures and localize faults that requires training with data from normal execution only.
- a prototype implementation of the two approaches
- a set of experimental results that evaluate the proposed approaches of *DyFAULT*.

This dissertation is organized as follows. Chapter 2 overviews the state-of-the art solutions, including mainstream approaches for failure prediction, error localization and error fixing, as well as a comparison between them. Chapter 3 introduces the design of the proposed solution, namely *DyFAULT*, and discusses how *DyFAULT* is composed of different techniques that may or may not require training with faulty executions. Chapter 4 presents the implementation of the two different workflows. Chapter 5 and Chapter 6 discuss the design and implementation of the two approaches for predicting failures and localizing faults. Chapter 7 discusses the methodology and design of the experiments that validate the proposed approach. Chapter 8 presents the experimental data that we obtained with the prototype implementation on a testbed, with a discussion of its validity. Chapter 9 summarizes the main results of my Ph.D research.

Chapter 2

Software Self-healing

In this chapter, I review the main state-of-the-art techniques for implementing the activities that comprise a self-healing approach: failure prediction, fault localization and error fixing.

Salfner et al.'s and Wong et al.'s surveys provide a comprehensive overview of the main approaches for predicting failures and localizing faults. Salfner et al. [106] survey approaches for predicting failures and identify four main classes of approaches depending on the required input data: failure tracking, symptom monitoring, detected error reporting, undetected error auditing approaches. *Failure Tracking* approaches predict failure occurrences based on past history of failures. *Symptom Monitoring* approaches predict failures by analyzing periodically measured system variables. *Detected Error Reporting* approaches predict failures by analyzing error reports, such as error logs, by exploiting rules, co-occurrence, error patterns, statistical tests or classifications. *Undetected Error Auditing* approaches search for undetected errors to predict future failures. Wong et al. [135] survey software fault localization techniques, and classify 385 publications about fault localization. They identify eight main classes of fault localization approaches: slice-based, spectrum-based, statistics-based, program state-based, machine learning-based, data mining-based, model-based, and miscellaneous. These surveys of approaches for predicting software failures and localizing faults provide an important background information for studying and developing failure prediction and fault localization solutions that cope with the unique characteristics of multi-tier system. In this chapter, I discuss in details techniques developed for multi-tier systems, and in particular cloud system, and highlight the limitations of current failure prediction and fault localization techniques for multi-tier systems, to identify the main motivations for my work.

2.1 Failure Prediction

Failure prediction approaches refer to techniques that determine whether a system will fail. They can be standalone services deployed for the system administrators [63], or part of a self-healing infrastructure [106]. We identify two main classes of approaches for predicting failures, *signature-based* and *anomaly based*.

Signature-based approaches encode failure-prone characteristics into signature models and predict failures by comparing signature models with runtime behaviors. *Anomaly-based approaches* learn behavioral models from non-failure-prone behaviors, capture deviations at runtime anomalies, and are often implemented as one-class anomaly detection techniques [24],

2.1.1 Signature-Based Approaches

Vilalta et al. [128] propose a data-mining approach that forecasts failures, called target events, by capturing predictive subsets of events, called eventsets, occurring prior to a target event. Vilalta et al.'s approach is composed of the three steps: 1. Identifying target events for the occurrence of types of events frequently preceding them within a certain time window: Eventsets that do not occur with a minimum probability before a target event are filtered out. 2. Validating frequent eventsets to ensure that the probability of an eventsets appearing before a target event is significantly higher than the probability of not appearing before target events. Eventsets that do not occur with a give probability before target events and often occurs before non-target events are discard. 3. Building a rule-based model that predicts failures from the validated eventsets.

hPREFECTs [45] models failure propagation phenomena by investigating failure correlations in both time and space domains to predict failures. hPREFECTs describes failure events and the associated performance variables as a formal representation, called failure signature, that allows hPREFECTS to cluster correlated variables to predict failures. Each compute node has a event sensor that tracks events recorded to the local event logs, extracts failure records and creates formatted failure reports for the failure predictor. It also monitors the performance dynamics of executing applications and measures the resource utilization. The failure predictor estimates future failure occurrences based on information collected with the event sensor. hPREFECTS uses a spherical covariance model with an adjustable timescale parameter to cluster failure signatures in the time domain aiming to quantify temporal correlation among failure events, and exploits an aggregate stochastic model to cluster failure signatures in the space domain and use these groups for failure prediction.

Similar Events Prediction (SEP) [107] is an error pattern recognition algorithm based on semi-Markov chain model and clustering. The model encodes both the time of error occurrences and the information contained in error messages (e.g. error type), called properties of the event. The time between errors is represented by the continuous-time state transition duration of the semi-Markov chain with uniform distributions, while each state in the semi-Markov chain encodes restrictions on properties of events and their position in the event pattern. Training the model involves hierarchical clustering of error sequences leading to a failure and computation of relative frequencies to estimate state transition probabilities. SEP predicts a failure if the failure probability exceeds a specified threshold.

Ozchelik and Yilmaz's *Seer* technique combines hardware and software monitoring to reduce the runtime overhead, which is particularly important in telecommunication systems [94]. *Seers* trains a set of classifiers by labeling the monitored data, such as caller-callee information and number of machine instructions executed in a function call, as passing or failing executions, and uses the classifiers to identify the signatures of incoming failures.

Nistor and Ravindranath's *SunCat* approach predicts performance problems in smartphone applications by identifying calling patterns of string getters that may cause performance problems for large inputs, by analyzing similar calling patterns for small inputs [91].

Malik et al. [81] developed an automated approach to detect performance deviations before they become critical problems. The approach collects performance counter variables, extracts performance signatures, and uses signatures to predict deviations. Malik et al. built signatures with a supervised and three unsupervised approaches, and provide experimental evidence that the supervised approach is more accurate than the unsupervised ones even with small and manageable subsets of performance counters.

2.1.2 Anomaly-Based Approaches

PREdictive Performance Anomaly pREvention (PREPARE) system [138] provides automatic performance anomaly prevention for virtualized cloud systems. PREPARE applies statistical learning algorithms on system metrics (e.g., CPU, memory, network I/O statistics) to early detect anomalies, and coarse-grained anomaly cause inference to identify faulty VMs and infer metrics that are related to performance anomalies. PREPARE consists of four major components: 1. VM monitoring tracks system metrics (e.g. CPU usage) of different VMs using a pre-defined sampling interval. 2. Anomaly prediction classify events with respect to future data to foresee whether the application will enter an anomaly state in a finite horizon. It uses a two-dependent Markov chain model to predict metric values, and a Tree-Augmented Naive (TAN) Bayesian network to classify normal or abnormal states and to rank metrics that are mostly related to an anomaly. 3. Anomaly cause inference filter alerts raised by the predictor with a majority voting scheme that discards false alarms based on the observation that most false alarms are caused by transient and sporadic resource spikes. If the alert is not a false positive, a fast diagnostic inference identifies both the faulty VMs and the metrics related to the predicted anomaly. 4. Predictive prevention actuation triggers the proper anomaly prevention actions according to the identified faulty VMs and relevant metrics.

Fulp et al.' spectrum-kernel SVM approach predicts disk failures using system log files [46]. Fulp et al. exploit the sequential nature of system messages, the message types and the message tags, to distill features that a SVM model processes to identify message sequences that deviate from the identified patterns as symptoms of incoming failures.

Guan et al. [101] use a learning approach based on Bayesian methods to predict failure dynamics in cloud systems. Guan et al.'s approach works in an unsupervised manner and deal with unlabeled datasets. A probabilistic model takes a measurement as input and outputs its probability of appearance as a normal behaviour. Guan et al.'s prototype implementation uses sysstat to collect runtime performance metrics from the Cloud hosts, and uses a mutual information-based feature selection algorithm to choose independent features that capture most information. Then, Guan et al. apply principal component analysis to reduce redundancy among the selected features, detect failures with an ensemble of Bayesian models that represent a multimodal probability distribution, and execute an Expectation-Maximization algorithm to determine the probability of failures.

ALERT introduces the notion of alert states, and exploits a triple-state multi-variant stream classification scheme to capture special alert states and generate warnings about incoming failures [119].

Tiresias integrates anomaly detection and Dispersion Frame Technique (DFT) to predict anomalies [133].

Sauvanau et al. capture symptoms of service level agreement violations: Sauvanau et al.'approach collects application-agnostic data, and classifies system behaviors as normal and anomalous with a Random Forest algorithm [109].

2.1.3 Loosely Related Approaches

Over the past decade software performance has become a complex problem in software engineering, and many researchers and professionals have developed several approaches to detect performance anomalies or regressions.

Performance anomaly detection approaches focus on finding performance issues in productive

systems, to identify performance bottlenecks [61].

The BARCA framework [31] detects and classifies anomalies in distributed systems, without requiring historical failure data. BARCA collects system metrics and extracts features such as mean, standard deviation, skewness and kurtosis, uses SVMs to detect anomalies, and applies a multi-class classifier to discriminate anomaly behaviors such as deadlock, livelock, unwanted synchronization, and memory leaks.

The Root tool [65] works as a Platform as-a-Service (PaaS) extension. Root detects performance anomalies in the application tier, and classifies their cause as either workload change or internal bottleneck. Root executes a weighted algorithm to determine the components that are most likely the bottleneck.

TaskInsight [143] is an anomaly detector for cloud applications based on clustering. TaskInsight detects and identifies application threads with abnormal behavior, by analyzing system level metrics such as CPU and memory utilization.

All these approaches differ from our work mainly in two directions: (i) they usually do not locate the faulty resource that causes the performance anomaly; (ii) they are not able to detect performance problems at different tiers. Detecting performance problems at different tiers, from high-level application service components to virtualization and hardware resources is an open challenge [61].

Performance regression detection approaches focus on detecting changes in software system performance during development with the ultimate goal of preventing performance degradation in the production system [26]. Performance regression detection approaches fundamentally seek to verify whether the overall performance of the development system has changed as a result of recent changes to the code.

Ghaith et al. [50] proposed transaction profiles as a measurement to detect performance regression. The profiles reflects the lower bound of the response time in a transaction under idle condition and do not depend on the workload. Comparing transaction profiles can reveal performance anomalies that can occur only if the application changes. Foo et al. [43] proposed an approach to automatically detect performance regressions in heterogeneous environments in the context of data centers. Foo et al.'s approach uses an ensemble of models to detect deviations in performance., and aggregate the deviations using either a simple voting algorithm or more advanced weighted algorithms to determine whether the current behavior really deviate from the expected one or whether it was simply due to an environment-specific variation. Unlike performance regression detection approaches that assume a variable system code base and a stable runtime environment, *PreMiSE* collects operational data to predict failures and localize faults caused by a variable production environment in an otherwise stable system code base.

2.2 Fault Localization

The many different error localization approaches implement few main analysis techniques. Several approaches exploit *data analysis* and *machine Learning* [135], some approaches exploit techniques that apply to the specific characteristics of large multi-tier systems, namely *Latency Analysis* and *Graph-based Algorithms*. Yet other approaches adapt techniques originally developer in other contexts, such as *Fault Diagnosis in Self-adaptive Systems*.

2.2.1 Latency Analysis

Latency analysis consists of identifying anomalous latency aspects at different granularity levels to diagnose possible faulty elements responsible for the anomalies.

CloudDiag [86] captures user requests, analyzes end-to-end tracing data (in particular, execution time of method invocations) and finds method calls that are likely responsible for an observed performance anomaly according to their latency distribution.

DARC [126] finds the root cause path, which is a call-path that starts from a given function and includes the largest latency contributors to a given peak, by means of runtime latency analysis of the call-graph.

Roy et al.propose an algorithm to identify network locations (e.g., routers) that are likely responsible for observed performance degradation by investigating latency data of a deployment [105].

Khanna et al. at Amazon provide an architecture to localize faulty node or links in the network topology by collecting network measurements such as packet latency [72].

2.2.2 Data Analysis

Data analytics approaches exploit various kinds of statistical techniques that include canonical correlation analysis, outlier detection, change detection statistics, probabilistic characterization and event-distribution analysis.

Canonical correlation analysis detects errors based on the change in correlations. Wu et al. propose a Virtual Network Diagnosis (VND) framework that extracts the relevant data from virtual network diagnosis [136]. PeerWatch [70] and FD4C [129] uses a statistical technique, canonical correlation analysis (CCA), to extract the correlated characteristics between multiple application instances.

Outlier detection detects errors exploiting standard deviations. PerfCompass [37] collects performance data about system calls while virtual machines behave correctly. When PeerCOmpass observes a performance problem, it analyzes the collected data and classifies the root cause of the error as either external or internal.

Change detection statistics identifies the frequency of changes in the probability distribution of a stochastic process or a time series. Johnsson et al. propose an algorithm to determine the root cause location (links) of a performance degradation by using measurements (e.g. delay or packet loss) and a graph model representing the network [67].

Probabilistic characterization of performance deviations exploit statistical techniques to build probabilistic characterizations of expected performance deviations. Shen et al. propose a reference-driven approach to monitor low level Linux events and to use the data collected during failures as reference to detect similar errors [114]. Draco [71] is a scalable engine to diagnose chronic problems by using a “top- down” approach that starts by heuristically identifying user interactions that are likely to have failed, such as dropped calls, and drills down to identify groups of properties that best explain the difference between failed and successful interactions by using a scalable Bayesian learner. Cherkasova et al. proposes a framework to collect system and JMX metrics, then distinguish performance anomalies, application transaction performance changes and workload change, which reduce the number of false positives in anomaly-based prediction and localization techniques [30]. Herodotou et al. focus on data center failures [57], using active monitoring such as ping to detect and localize errors, producing a ranked list of devices and links that are related to the root cause.

Event-distribution analysis identifies anomalous distributions of events in space or time. Pecchia et al. propose a heuristic to filter out useless and redundant logs to improve the precision of log analysis in failure diagnosis [96].

2.2.3 Machine Learning

Many approaches exploit different kinds of machine learning techniques.

Statistical machine learning approaches exploit a statistical model, often parameterized by a set of probabilities. Clustering approaches use unsupervised clustering to determine operational regions where the system operates normally to identify faulty ones. DeepDive [92] identifies signals if a VM may suffer interference problems by monitoring and clustering low-level metrics such as hardware performance counters. If DeepDrive suspects the existence of some interference problems, it compares the metrics produced by the VM running in production and in isolation to confirm the interference. UBL [35] leverages on Self-Organizing Maps (SOM), a particular type of Artificial Neural Networks (ANN), to capture emergent behaviors and unknown anomalies.

Pinpoint [27] is a framework for root cause analysis on the J2EE platform that requires no knowledge of the application components. It diagnoses the root cause component of an error by tracing and clustering requests. Duan et al. propose an algorithm to label unknown instances (represented by metric data) with litter human interaction [40]. The unknown instances can be faulty or normal and the algorithm is able to classify most of the instances automatically by inquiring sysadmin and applying learning techniques.

Classification approaches use supervised classification to identify the errors. Yuan et al. use a typical machine learning approach that analyzes system call information (on Windows XP), and classifies the information according to known errors [139]. Zhang et al. propose an approach that can update a metric attribution model according to the change of the workloads and external disturbances [142].

Symbolic machine learning approaches explicitly analyze symbolic models to localize errors according to new observed data. POD-Diagnosis [137] deals with errors in sporadic operations (e.g., upgrade, replacement and so on) by extracting events from logs, detecting errors from events, and using a fault tree to identify the root causes. BCT [82] exploits behavioral models inferred from tests execution to identify the causes of software failures.

2.2.4 Graph-based Algorithms

Graph-based approaches localize errors by using graph models derived from some system characteristics like network topology and service dependencies. Sherlock [142] diagnoses faulty source, such as components, in enterprise networks by monitoring the response time of services. Based on the Inference Graph Model, Sherlock develops an algorithm called Ferret to localize the problem.

NetMedic [69] works on enterprise networks to perform detailed diagnosis with minimum application knowledge. NetMedic mainly uses automatic dependency graph generation and historic data reasoning. Sharma et al. propose an algorithm to perform error localization (problematic metrics) by using a dependency graph that is built with system metrics [112].

Gestalt [90] combines the features of existing algorithms. Gestalt takes transaction fails as inputs and finds the culprit component. Cluster-MAX-COVERAGE (CMC) [120] is a greedy algorithm to diagnose the large-scale failures in computer networks. Moreover, an adaptive

algorithm is also proposed as a hybrid error diagnosis approach to determine whether a failure is independent or clustered.

2.2.5 Fault Diagnosis in Self-adaptive Systems

Self-Adaptive Systems are systems that adjust their behavior in response to changes in the environment and in the system itself [28]. Techniques for diagnosing faults are a fundamental element in self-adaptive systems. Casanova et al. [21] and Abreu et al. [1] propose Spectrum-based Multiple Fault Localization (SMFL), a technique that models the system behaviors with an expressive language to decide the amount of data needed to diagnose faults, in the form of entropy values. Casanova et al. combine SMFL with architecture models for monitoring system behavior, to apply the approach to larger variety of systems [19], extend the approach to localize faults is system components that are not directly observable, and propose formal criteria to determine if a given information is enough to accurately localize faults [20].

2.2.6 Hybrid Approaches

Some approaches combine different techniques. PerfScope [36] combines hierarchical clustering, outlier detection and finite state machine based matching, Kahua [118] combines clustering and latency analysis, CloudPD [113] combines Hidden Markov Models (HMM), k-nearest-neighbor (kNN), and k-means clustering to build the performance model, and uses statistical correlations to identify anomalies and signature-based classification and identify errors, Pingmesh [55] combines data analytics and latency analysis.

2.3 Error Fixing

Techniques for error fixing vary depending on granularity, platform environment and running services. As an example, modern software design takes into consideration dividing a giant running application into micro services, and carefully separating stateful components and stateless components. For stateless components, simply tearing down a running service and bringing up a substitution generates no harm to the expected behavior, while for stateful parts, keeping track of the state for restoring in case of a service down is necessary. Additionally, software and application redundancy provides support for service availability during switching problematic services to fixed ones.

We classified techniques and tools for error fixing into three categories, software aging prevention, redundancy and runtime state restore.

2.3.1 Software aging prevention

Long-running and computation-intensive applications suffer particularly from *age*. Non-deterministic faults can lead to system crashes, but often degrade the system performance because of memory leaks, memory caching, weak memory reuse, etc, and lead to system failures only after a long time [59].

Software rejuvenation periodically restarts an application to clean the environment and re-initialize the memory and the data structures, thus preventing the system to fail because of age [59]. The runtime costs of rejuvenation may be high. For example restarting a Web server

may result in a service downtime, or restarting a transactional system may cause losses of the current transaction incurring in rollback costs [54]. Thus, rejuvenating the system at the right time is important. A strategy to select the optimal interval is to measure some system attributes that show the symptoms of aging [53].

Instead of restarting the whole application, *Micro-Rebooting* reboots only the system components that failed or show aging signs. Micro-reboot consists in individual rebooting of fine-grain application components. It can achieve many of the same benefits as whole-process restarts, but an order of magnitude faster and with an order of magnitude less lost work. [13].

Environment Changes deal with software aging by re-executing failing programs under a modified environment [102].

2.3.2 Redundancy

Software Redundancy are in commonly use to guarantee service reliability. Differently from mirroring, redundancy usually features components that are functionally equal but diverse in implementation, that prevents back-up redundant service from falling into the same failing point.

N-Version Programming improves fault tolerance by independently designing multiple functionally-equivalent programs, and by comparing the results of the different programs to identify single faults [23, 39, 48, 58, 80].

Recovery Blocks are redundant implementations that substitute faulty implementations in the presence of runtime failures [5, 6, 12, 38, 87, 88, 108, 116, 117].

Workarounds exploit redundant code to heal from failures by substituting failing blocks. The faulty element is usually replaced with a redundant one. This may mean that the faulty instance is killed, possible still allocated resources are freed, and a new instance is started [14, 15, 16, 17, 18, 42, 51, 75, 89].

Diversity approaches switch to different elements to solve the same task [29, 34, 85].

Redirection approaches change the task-flow in the presence of a failure to a recovery routine, and then go back to the original task-flow [25, 125].

2.3.3 Runtime state restore

Some applications are state specific, which complicates the determinism of software behavior by different user context and environments. Industrial applications usually restore states by predefined behavior specification, or backing up critical states.

Forcing acceptable behavior approaches fix faults by forcing the system to behave as expected, to reach the correct goal [32, 49, 56, 95, 98, 131].

Evolution exploits genetic programming, a problem solving approach inspired from natural evolution to derive a correct program that fixes the fault from a failing program. [2, 68, 78, 132]. Genetic programming uses intrinsic redundancy to produce variants of a failing program, and selects the variants that appear to be more likely correct. Such variants are progressively refined and evolved in a process of mutations and re-combinations, until reaching either a solution or an acceptable approximation of it. To evaluate whether a variant is evolving towards an acceptable solution or must be discarded, each variant is measured with a *fitness function* that determines if a variant improves or not. Those variants that behave well are chosen to breed, and to evolve into a new set of programs that are a step closer to the solution. There are two main strategies to create new programs, called genetic operations: *crossover* and *mutation*. The former creates

a child by combining parts of code of two selected programs, the latter, by choosing a code fragment of a program and altering it following a set of predefined rules. Researchers applied genetic programming in the context of automated fault fixing.

Checkpoint and Recovery safely recover and re-execute software systems to recover from runtime failures, and can be used also to increase the performance of rejuvenation [41, 47, 130, 140].

Isolation approaches cut off failing elements to assure that its malicious behavior does not infect the other parts of the system [1].

Migration approaches move a failing instance from a host to a different host, redirecting also the possibly affected communication and data flow [33, 79, 115].

2.4 Summary

Section 2.1 presents several types of approaches for predicting failures. *Signature-based* approaches can accurately predict failures, but fails to capture unseen ones. *Anomaly-based* approaches can handle a wide range of types of failures, while it often produces false alarms. The challenge of failure prediction is to have a trade-off between the two approaches.

Section 2.2 discusses different categories of fault localization solutions. Despite the variety of solutions, some of them are restricted by the platform, some of them only works on specific fault types, and some of them are limited by the granularity. Therefore, an approach targeting on IaaS level distributed cloud system which fits a wide range of faults is necessary.

Section 2.3 provides state-of-the-art solutions for error fixing. Due to the complexity and abundance of the error fixing, this thesis mainly presents solutions for failure prediction and fault localization, and leaves the selection of error fixing techniques to the administrators according to the actual requirements.

State-of-the-art solutions for predicting failures and localizing faults work well for specific systems, but do not generalize to large multi-tier complex systems. In particular, state-of-the-art solutions for predicting failures, localizing faults and self-healing solutions for multi-tier complex systems face the following challenges: (i) many of them do not work with the strict constraints imposed in commercial multi-tier systems, in particular, the limitations of seeding faults for training, (ii) some of them are validated with benchmark data collected for specific usages, such as load test, and do not work with data that reflect real workload dynamics and variety, (iii) most of them target specific fault types, and address only a subset of real world multi-tier system faults. (iv) most of them do not achieve both high precision and recall at the same time.

DyFAULT addresses these challenges by defining and developing failure prediction and fault localization solutions for target systems that correspond to industrial cases and within industrial constraints. We studied and validated the *DyFAULT* solutions referring to a system implementation on commercial off-the-shelf hardware on top of common open source solutions for implementing multi-tier systems. We select services and shape the workload based on documentation and suggestion from our industrial partners, and focus on a set of types of faults most frequently observed in real world large multi-tier systems' that we retrieved from bug repositories and failure cases. *DyFAULT* also applies a two layer data processing model to improve the precision and recall of failure prediction.

[1] http://www.oracle.com/us/products/servers-storage/solaris/solaris-pred-self-healing-ds-075587.pdf

Chapter 3

DyFAULT

In this chapter, I present *DyFAULT* (Dynamic FAULT Localization), a new approach for predicting failure and localizing faults. I start with a behavior description of *DyFAULT* in Section 3.1, then I present the design of *DyFAULT* in Section 3.2. In Section 3.3, I discuss the two workflows of *DyFAULT*, namely *PreMiSE* and *LOUD*, respectively.

3.1 The DyFAULT framework

Figure 3.1 illustrates the overall *DyFAULT* dataflow. *DyFAULT* periodically collects system runtime metric data. The tables in Figure 3.1 are an excerpt of the collected metrics that include *CPU usage percentage on Machine 2*, *network latency in the mail service*, and *number of packets sent on Virtual Machine 1*. *DyFAULT*'s *Anomaly Detector* analyzes the runtime metric data. In the example of Figure 3.1, at time 0, the *Anomaly Detector* observes that the value of *CPU usage percentage on Machine 2* goes beyond 90, which deviates from its normal range in the past, thus regard it as an *anomaly*(marked in red in the table). At time 2, the *Anomaly Detector* finds three anomalies.

A single deviation of a value can be just a false alarm. In this example, at time 0, the anomaly *<Machine 2, CPU Usage, 80>* may come from a temporary system check that consume a significant CPU resource in a very short time, which will not result in a failure. Therefore, the outcomes of the *Anomaly Detector* is not evident enough for a prediction or localization.

DyFAULT's *Anomaly Processor* refines and analyzes the anomalies, taking into account the interactions between the metrics. In Figure 3.1, at 0, the *Anomaly Processor* consider a single anomaly not significant enough to lead to a potential failure. While at time 2, the three anomalies indicate a propagation of errors among the metrics. The *Anomaly Processor* generates a failure alert and infers that the fault is most likely a *CPU hog on Virtual Machine 1 hosted on Machine 2*.

In my thesis I focus on the design and implementation of *DyFAULT* to producing failure alert, fault type and location.

3.2 Framework Overview

Figure 3.2 shows the main components of *DyFAULT*: monitored data, anomaly detector, intermediate data, anomaly processor, alert information.

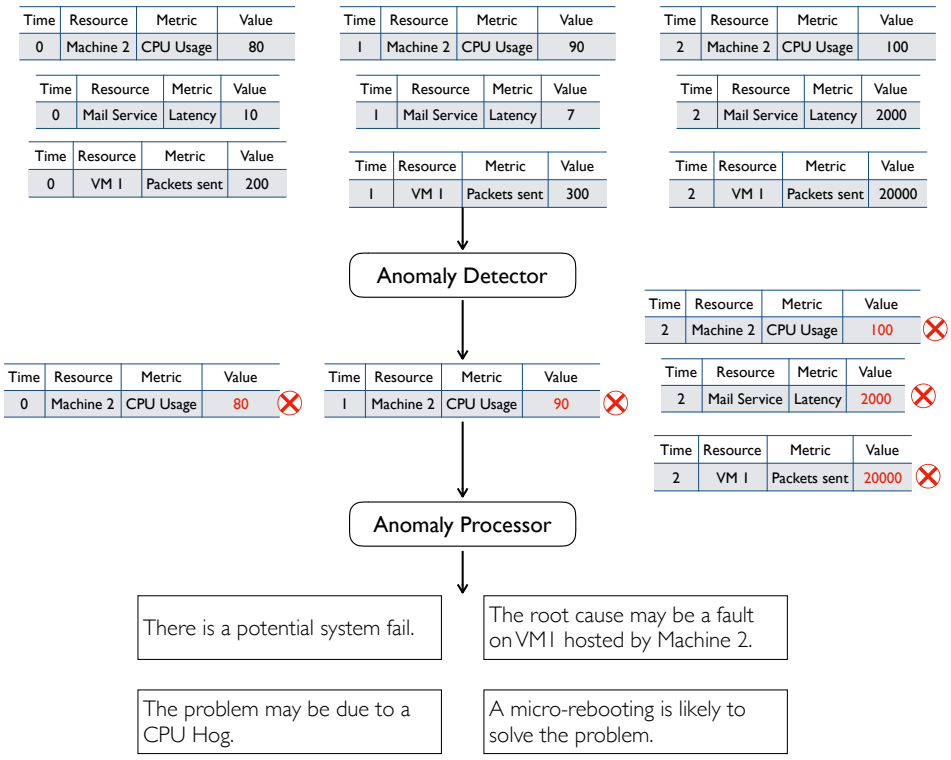

Figure 3.1. The DyFAULT freamework

3.2.1 Monitored Data

The *Monitored Data* are Key Performance Indicators, KPI, that is, $< resource, metrics >$ pairs. For instance, a *KPI* can be the CPU Usage of a specific virtual machine, or the service latency of a specific application. A KPI specifies a measurable unit of the running system.

At runtime, *DyFAULT* monitors KPI data from the system. At different timestamp, a KPI may have different values. The input of *DyFAULT* is a series of "KPI-Value" tuples (as shown in Figure 3.1).

DyFAULT uses input data to train models (*training data*) that *DyFAULT* uses to predict failures on new input data (*operational data*).

3.2.2 Anomaly Detector

The *DyFAULT Anomaly Detector* processes the monitored data at training time to build models and at operational time to detect anomalies.

The *DyFAULT Anomaly Detector* (i) identifies anomalous values, (ii) detects correlations between KPIs, and (iii) produces a graph data structure to represent the pairwise correlations for further analysis.

The Anomaly Detector is composed of a *Baseline Model Learner* that analyzes historical data, involving *Data Analytics* technique, such as statistical inference, to generate the *Intermediate*

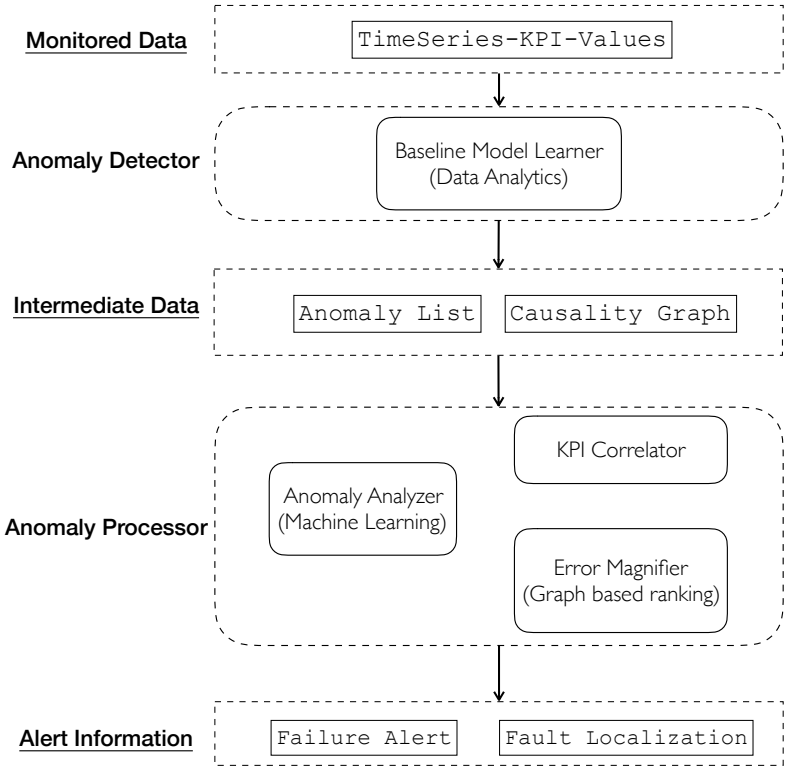

Figure 3.2. Overall design

Data.

3.2.3 Intermediate Data

The Anomaly Detector produces *Intermediate Data* that include an *Anomaly List* and a *Causality Graph*.

The Anomaly List is a list of anomalous KPIs values at a given timestamp.

The Causality Graph is a graph that models the correlation between KPIs. Each vertex of the Causality Graph represents a KPI, and the weights of the edges indicate the correlation significance between the KPIs that correspond to the nodes.

DyFAULT uses the Anomaly List as the regular input of the Anomaly Processor, and the Causality Graph as *shared knowledge* that the *Anomaly Processor* uses on demand.

3.2.4 Anomaly Processor

The *Anomaly Processor* refines the intermediate data, and is composed of an Anomaly Analyzer , a KPI Correlator and an Error Magnifier .

The *Anomaly Analyzer* analyzes the anomaly list. The anomaly list is a vector of boolean values that indicate the KPIs with anomalous values at the given timestamp. The *Anomaly*

Analyzer deals with the Anomaly List as a classification problems given vector-like inputs.

The *KPI Correlator* and *Error Magnifier* combine the Anomaly List and the Causality Graph for further analysis. The *KPI Correlator* marks anomalous KPIs in the Causality Graph, and removes the KPIs with normal valuers to produce a graph of anomalous KPIs. The *Error Magnifier* implements different graph-based ranking algorithms to rank the significance of the anomalous KPIs and infer likely root cause of the fault.

3.2.5 Alert Information

DyFAULT coordinates the components in the *Anomaly Processor*, to issue *Alert Informations* that include: (i) an alert prediction: a boolean value that indicates if the monitored KPIs values suggests a potential failure' (ii) an alert type: the type of fault that will lead to the failure, for example a memory leak. (iii) an alert location: the subset of resources where the fault is most likely located.

An alert prediction constitutes a *Failure Alert*, and alert type and an alert location comprise a *Fault Localization*.

3.3 Execution based workflow

DyFAULT implements two main workflows, *PreMiSE* and *LOUD*, depending on the available training data that can be *Normal execution data* that are collected during normal executions, and *Faulty execution data* that are collected in the presence of active faults that lead to failures. While *Normal execution data* can be monitored on all target systems, collecting a useful amount of *Faulty execution data* requires intensive fault seeding that may not be always allowed, depending on the target applications.

PreMiSE requires both normal and faulty execution data, and process the training data in two stages to build the model, while *LOUD* trains models with normal training data only.

PreMiSE and *LOUD* share the anomaly detector component, but differ in the anomaly processor components, due to the different intermediate data that *PreMiSE* produces with both normal and faulty execution data, while *LOUD* produces with normal execution data only.

3.3.1 PreMiSE

Figure 3.3 indicates the *PreMiSE* workflow. The red and blue lines indicate the training workflow, while the black lines indicate the operational workflow.

PreMiSE iterates the model training twice, with normal execution data first and with faulty execution data later. *PreMiSE* trains the *Baseline Model Learner* with Normal Execution data, the red line in Figure 3.3, to build a model of normal behavior, and use the faulty execution data, the blue lines in Figure 3.3, to update the model with information about anomalies and failures.

At operational time *PreMiSE* executes the *Baseline Model Learner* with the model built with the first stage training to produce an Anomaly List, and executes the *Anomaly Analyzer* with the model trained from the second stage training to produce fault information, the black lines in the figure.

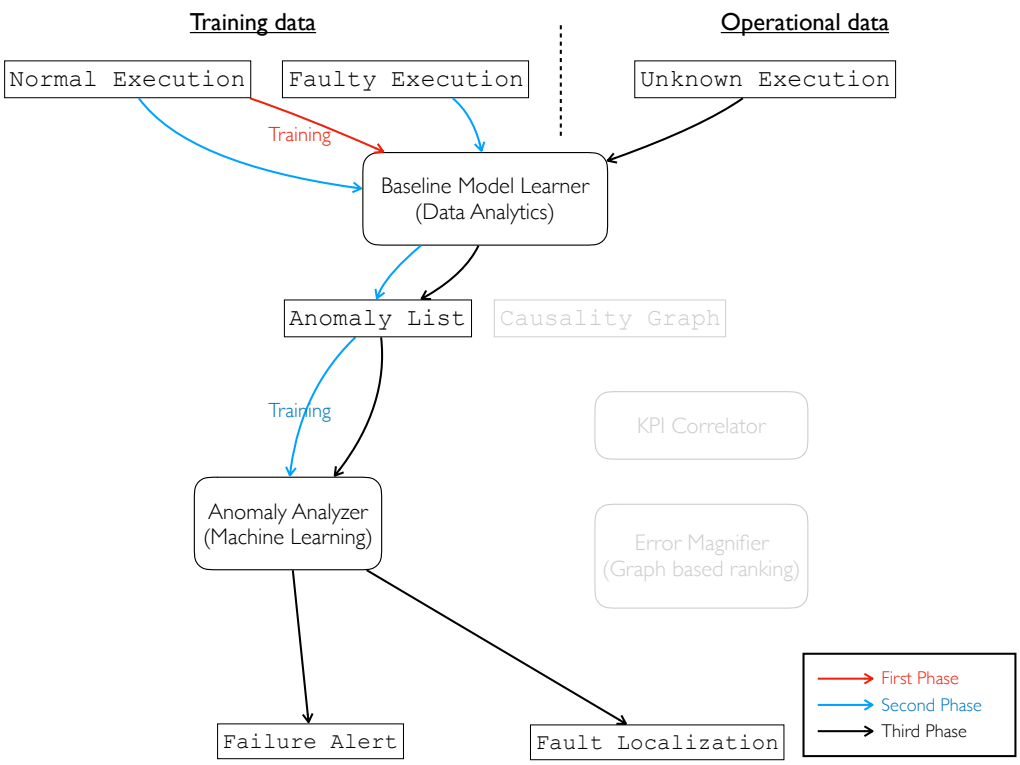

Figure 3.3. PreMiSE workflow

3.3.2 LOUD

Figure 3.4 shows the *LOUD* workflow. *LOUD* trains the models with normal execution data only, and proceeds in two training stages.

LOUD shares the first training stage with *PreMiSE*, as the *Baseline Model Learner* building the same model with KPI values collected only during normal execution. The *LOUD* anomaly list does not encode data about faulty executions, and the *LOUD* anomaly analyzer can predict failures, but not identify the fault type and location.

The *LOUD KPI Correlator* and *Error Magnifier* use the causality graph to rank the anomalies in the anomaly list according to their mutual correlations, to locate faults. The *LOUD KPI Correlator* prunes uncorrelated anomalies from the causality graph, while the *Error Magnifier* ranks KPIs according to the weighted connections in the pruned causality graph. *LOUD* identifies as possible fault location the resources of the highly ranked KPIs.

3.4 Summary

Current self-healing techniques do not cope with the complexity of data analysis of metric data of large multi-tier systems, complexity that is due to a large amount of KPIs and a high dynamics of value range for each KPI.

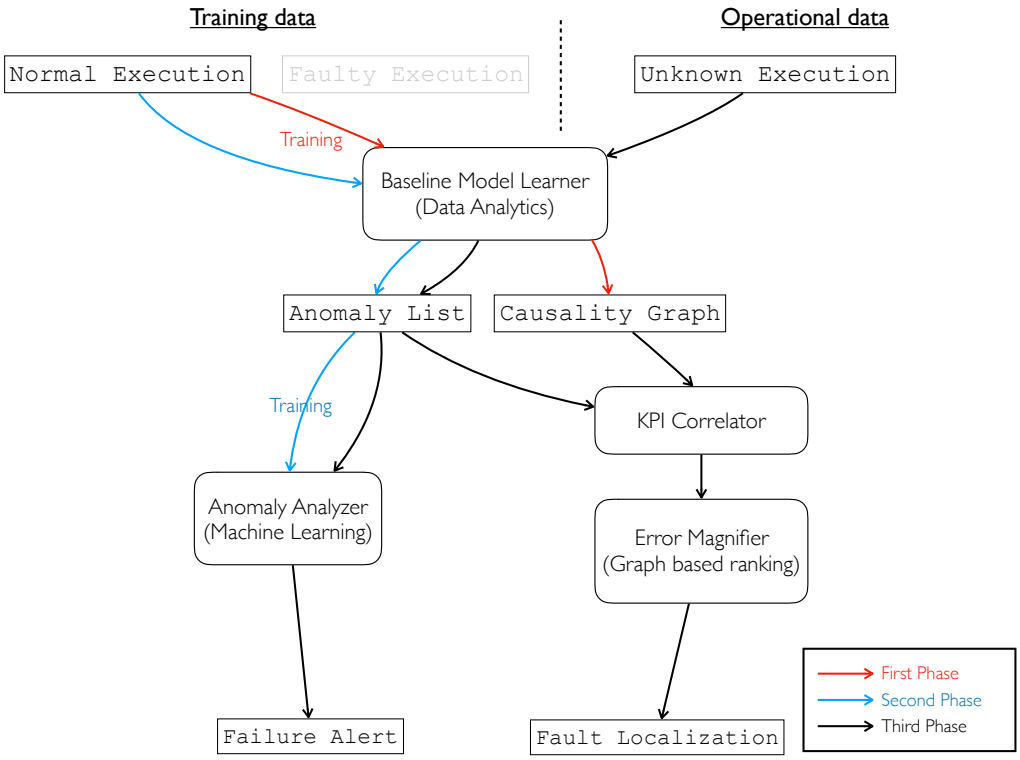

Figure 3.4. LOUD workflow

DyFAULT reduce the high dimension of raw input KPI data by separating failure prediction and fault localization into two stages, to reduce the high dimension of raw input KPI data. The first stage focuses on single KPIs, and transform the corresponding data from concrete numeric value to discrete binary value that indicates whether or not a single KPI is normal. The second stage aggregates all KPIs and consider their interaction to produce failure prediction and fault localization, benefiting from the dimension reduction by the first stage, the second stage deals with reasonable complexity in the data.

Chapter 4

Anomaly Detection

In this chapter, I present the design and implementation of the *Baseline Model Learner* component, the component that *PreMiSE* and *LOUD* share to detect anomalies.

The *Baseline Model Learner* works in two stages, *training* and *operational*. At training time, *Baseline Model Learner* leverages KPIs values monitored during normal execution to learn a baseline model that captures the normal range of value for each KPI, and to produce a *Causality Graph* that captures the correlation between KPIs. At operational time, *Baseline Model Learner* analyzes KPI values with the trained model to detect anomalies. *Baseline Model Learner* groups KPIs with anomalous values at the same time into *Anomaly List*. The Anomaly List and Causality Graph form the intermediate data for further analysis.

Section 4.1 discusses the training of the *Baseline Model Learner* by presenting how the *Baseline Model Learner* samples data from the running system, and builds a model from the data sampled during normal execution. Section 4.2 describes the online operational phase of the *Baseline Model Learner* by discussing how the *Baseline Model Learner* detect anomalous KPI with the model.

4.1 Training

As shown in Figure 4.1, in the training phase, the *Baseline Model Learner* (i) monitors KPI data under normal execution of the system, (ii) builds a *Baseline Model* with the monitored data, and (iii) derives a *Causality Graph* that represents the causality relationship between KPIs.

4.1.1 KPI monitoring

DyFAULT elaborates KPI values periodically sampled with probes at different locations and at multiple layers in the system.

System administrators can use different probes, for instance general data collection tools [22], or application specific monitoring services [123]. Administrators can also configured the data sampling rate in their implementation according to requirement difference. In our experiments we collected over 600 KPIs of over 90 types, as discussed in Chapter 7.

DyFAULT elaborates KPI values on an independent machine, thus limiting the overhead on the target cloud system to the probes that impact on host machine's resource only with monitored data query and network communication, with negligible costs.

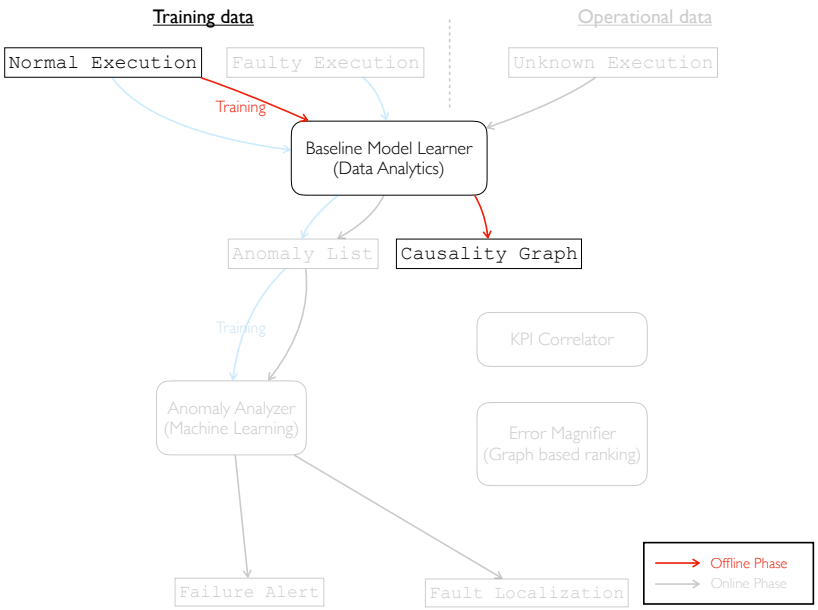

Figure 4.1. Offline Training of Baseline Model Learner

Table 4.1 presents a sample set of input data. Input data are time series data in the form of streams ordered by time. In the table, the specific KPI is $\langle Homer, BytesSentPerSec \rangle$. *Homer* is one of the virtual machines that we use in our experiments, a standard XML Document Management Server that stores MMTEL (MultiMedia TELephony) service settings of the users (see Chapter 7). *BytesSentPerSec* reports the outgoing number of bytes per second in the network communication. As the timestamp (first column) changes, sampled values(third column) also changes.

4.1.2 Baseline Model Learner

The *Baseline Model Learner* analyzes time series KPI data and derives the *Baseline Model* that models the behavior of the system under normal execution conditions. For each KPI, the model includes descriptive metrics of the historical data, such as mean value, maximum and minimum values, seasonality and seasonal offsets [10].

The *Baseline Model Learner* trains the models with raw data collected from KPI monitoring during a normal execution, and derives descriptive metrics from the input data with statistical test. The portability of the *Baseline Model Learner* relies on the flexibility of the approach: many algorithms that reads KPI data from the system and produces a model to judge whether an input is normal or not can be valid implementations of the detector. of *Baseline Model Learner* .

Figure 4.2 shows a baseline model of the KPI $\langle Homer, BytesSentPerSec \rangle$, as we discussed in the example in Section 4.1.1. The green area in the figure marks the range of values that the Baseline Model considers as normal as the *Baseline Model Learner* infers from the input data.

Baseline Model Learner infers also KPI correlation and models them in the form of a Causality Graph, For each pair of KPIs, *Baseline Model Learner* utilizes Granger causality tests [3] to mine

Table 4.1. Sample time series for KPI BytesSentPerSec collected at node Homer

Timestamp	Resource	BytesSentPerSec
...	...	...
Dec. 20, 2018 22:22:35	Homer	101376
Dec. 20, 2018 22:23:36	Homer	121580
Dec. 20, 2018 22:24:36	Homer	124662
Dec. 20, 2018 22:25:36	Homer	106854
...	...	...

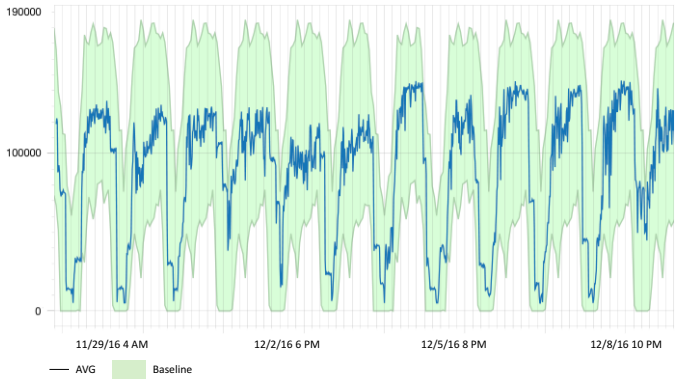

Figure 4.2. Sample baseline model of single KPI: BytesSentPerSec for Homer virtual machine

the impact between KPIs. A time series x is a *Granger cause* of a time series y, if and only if the regression analysis of y based on past values of both y and x is statistically more accurate than the regression analysis of y based on past values of y only.

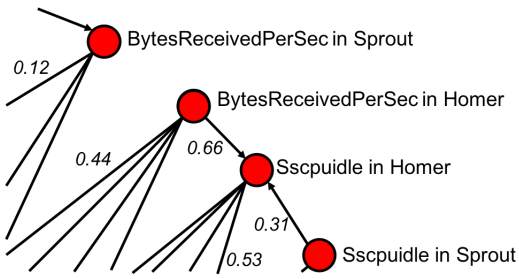

Figure 4.3. A sample baseline model: an excerpt from a Granger causality graph

Baseline Model Learner integrates the pairwise causality relationships and represents them as a graph, the *Causality Graph*. In the Causality Graph, vertexes correspond to KPIs and weighted edges indicate the significance of the causality relationship. The Causality Graph both assists the *Baseline Model Learner* in deriving the Baseline Model, and serves as a extra information in

fault localization, which we will discuss in Chapter 6.

Figure 4.3 shows an excerpt of a Granger Causality Graph. The vertexes correspond to KPIs, and the weights of the edges, ranging in the interval $[0, 1]$, indicate the significance of the causal relationship. For example in the figure, the KPI *BytesReceivedPerSec* in *Homer* has is strongly correlated with *Sscpuidle* in *Homer*.

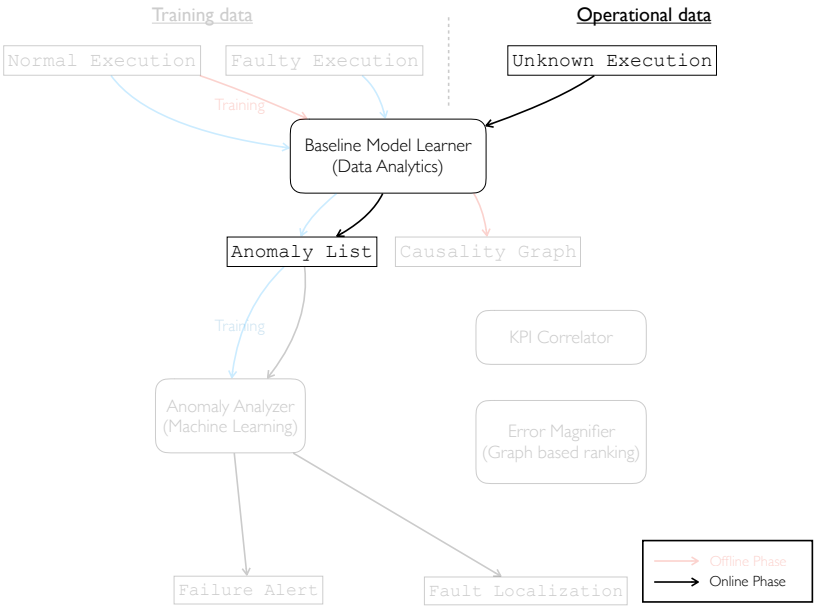

Figure 4.4. The operational time detection workflow of Baseline Model Learner

4.2 Detecting Anomalies at Operational Time

At operational time, *Baseline Model Learner* collects KPI monitored data, and uses the *Baseline Model* to identify anomalous KPIs (**anomalies**).

Figure 4.4 illustrates the workflow of the detection at operational time.

The *Baseline Model Learner* detects univariate anomalies as samples out of range, as shown in Figure 4.5. Given an observed value y_t of a time series y at time t, and the corresponding expected value $\hat{y}_t$ in y, y_t is anomalous if the variance $\hat{\sigma}^2(y_t, \hat{y}_t)$ is above an inferred threshold.

For each individual KPI, *Baseline Model Learner* performs the same analysis. Thus, at a given timestamp, *Baseline Model Learner* produces a list, which contains the KPIs that it detects as anomalous. Table 4.2 shows a sample of the anomaly list.

4.3 Summary

The *Baseline Model Learner* is the mounting point where *DyFAULT* gets the status of the monitored system's runtime and perform the initial analysis. *DyFAULT* uses system metrics in the form of time series data as its input, due to its lightweightness and portability.

Table 4.2. Sample the Anomaly List

Timestamp	Anomalies
...	...
Dec. 20, 2016 22:22:35	Null
Dec. 20, 2016 22:23:36	<Homer, BytesSentPerSec>
Dec. 20, 2016 22:24:36	<Homer, BytesSentPerSec>, <Sprout, BytesReceivedPerSec>, <Homer, MemoryUsage>, ...
Dec. 20, 2016 22:25:36	<Homer, BytesSentPerSec>, <Sprout, BytesReceivedPerSec>, <Homer, MemoryUsage>, ...
...	...

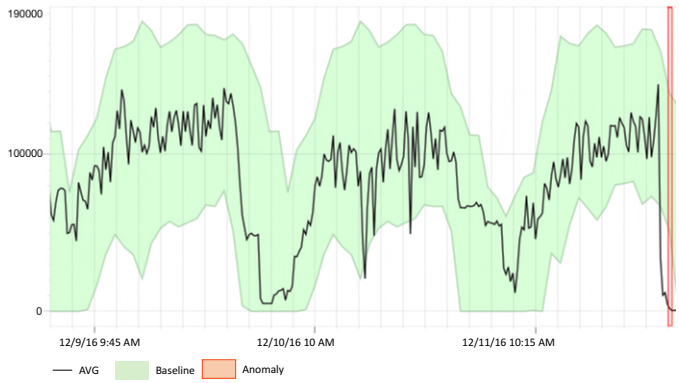

Figure 4.5. A sample univariate anomalous behavior

Because of the large number of system KPIs that a multi-tier system can provide in reality, it is challenging to build a unified model that is complicated enough to capture the variance of concrete numeric values of all KPIs. As covered by Section 4.1, for each KPI, *DyFAULT* , builds a single model that describes the normal statistic metrics of its value.

As covered by Section 4.2, at each timestamp during operational time, *Baseline Model Learner* reports each KPI as either normal or anomalous. A single anomalous KPI is not sufficient to reveal a potential failure in a large multi-tier system due to its complexity, thus further data processing is needed.

The set of anomalous KPIs, namely *anomalies*, along with the *Causality Graph* that models the correlation between KPIs, are the outcome of *Baseline Model Learner* and are used by *PreMiSE* and *LOUD*.

Chapter 5

PreMiSE

In this chapter, I present the design and implementation of *PreMiSE* , the component that detects and locates faults by relying on extensive training with data from both normal and faulty executions.

PreMiSE works in three phases: fault seeding, signature extraction at training time, failure prediction at operational time.

Section 5.1 discusses how *PreMiSE* exploits fault injection techniques to both observe failures and collect KPIs in the presence of failing executions for model training, and to study the ability of *PreMiSE* to predcit failures and locate faults. Section 5.2 describes how *PreMiSE* sets up the model for predicting failures by exploiting KPI values monitored when executing the target system while injecting faults. Section 5.3 presents the *PreMiSE* workflow for predicting failures during system operation.

5.1 Fault Seeding

In *PreMiSE*, we seed faults to explore diversity in faulty executions, and capture the system behavior in the presence of incorrect system states.

PreMiSE uses *Fault seeders* to inject faults of different types in different locations, and measures the system behavior under such different executions, to provide the *Anomaly Analyzer* with data to recognize fault type and location.

5.2 Signature Model Extraction

Figure 5.1 shows the workflow of the *PreMiSE* Signature Model Extraction, which is composed of a *Baseline Model Learner* and a *Anomaly Analyzer* that train the models that *PreMiSE* uses at operational time to reveal anomalies and predict failures.

The *Baseline Model Learner* trained with both normal and faulty executions produces models that can associate tuples of anomalies to specific fault types and locations. The *Baseline Model Learner* produces anomaly lists, **instance**, as vectors:

$$a_1, a_2, \ldots, a_n, \quad (T_{fault}, R_{fault})$$

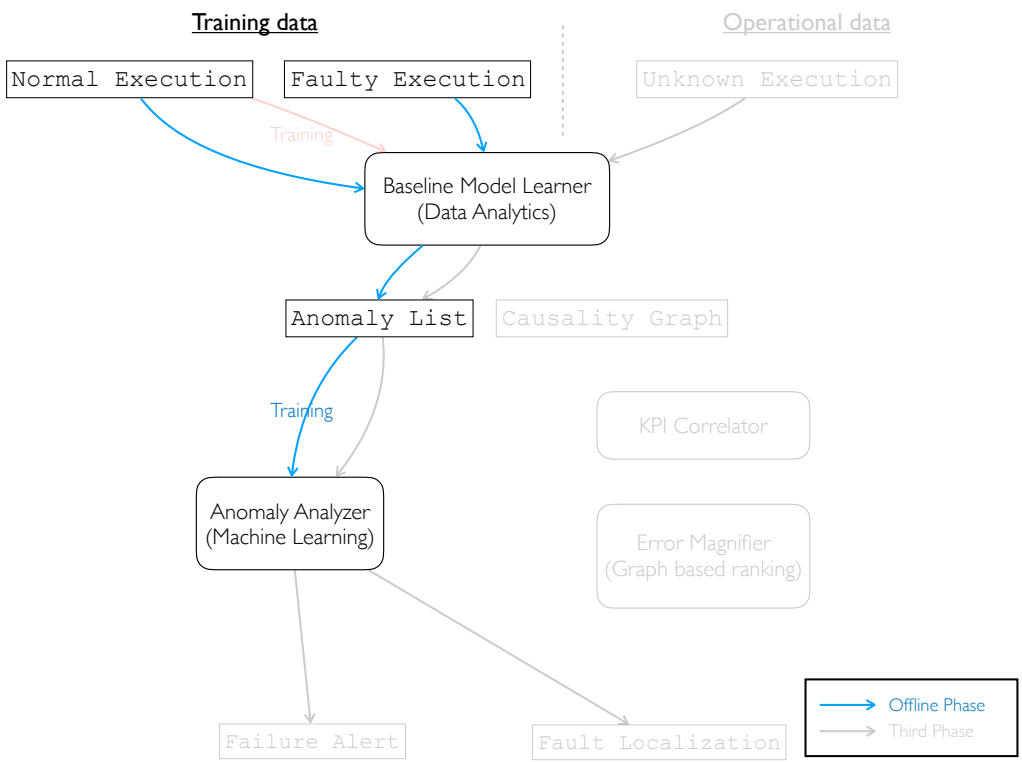

Figure 5.1. Workflow of Signature Model Extractor in PreMiSE

where a_i corresponds to anomalous KPIs that can continuous or discrete values, and the pair $\langle T_{fault}, R_{fault} \rangle$ indicate the fault type and location (resource) of the possible fault.

The *Anomaly Analyzer* processes time series of instances to learn pattern of anomalies, i.e. the *signatures* of different fault states. The *Anomaly Analyzer* uses machine learning algorithms to associate **labels** (tuples $\langle T_{fault}, R_{fault} \rangle$) to inputs (the rest of the vector). The *Anomaly Analyzer* employs *supervised machine learning* techniques to generate Failure Alert and Fault Localization. Supervised machine learning contains a set of algorithms, most of which use preliminary labeled data to calibrate their models for online prediction.

Figure 5.2 illustrates how *Anomaly Analyzer* determines fault type and location with a typical supervised machine learning algorithm, **K-Nearest Neighbors Algorithm**.

For simplicity in the figure, we assume only two KPIs, *CPU Usage* in resource *Homer*, and *Packet Retransmission* in resource *Homer*, and two types of faults, CPU hog and Packet loss, the resource *Homer*. The figure plots the anomalous values of each instance on the coordinate, and determines the category of an unknown instance, the "target" point in the figure, with the K-nearest Neighbors Algorithm that picks the K points closest to the target according to the Euclidean distance (here we set $K = 6$), and infers the fault type and location form the majority of labels associated to these K points. In the figure, five labels belongs class *Packet Loss in Homer* and one label belongs to *Normal*, thus the *Anomaly Analyzer* labels the target instance as *Packet Loss in Homer*.

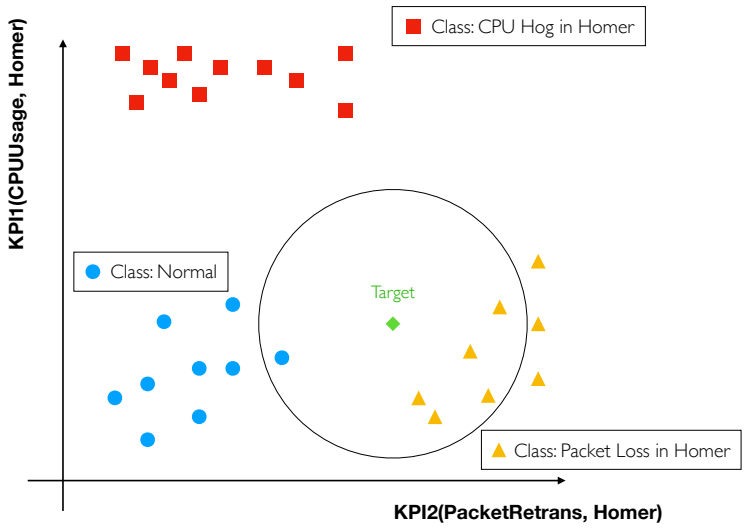

Figure 5.2. A sample signature model based on K-nearest neighbors algorithm

The K-Nearest Neighbors algorithm is an approximated solution of the problem:

- The importance of KPIs in expressing the fault information differs, while Euclidean distance will just ignore the variety of KPIs.
- The value of KPI anomaly, i.e., a_i, is usually a bivariate value that indicates whether or not a KPI is anomalous, thus introducing a lot of overlapping on the coordinate.
- The number of instances for each type of fault can be different, thus impacting on the fairness of selecting the K nearest neighbors.

Therefore, it is important to carefully select proper supervised machine learning algorithms to handle failure prediction and fault localization problem. We empirically investigated signature extractors based on several algorithms:

- function-based *Support Vector Machines (SVM)* that implement a sequential minimal optimization [99],
- *Bayesian Networks (BN)* based on hill climbing [52],
- best-first *Best-First Decision Trees (BFDT)* that build a decision tree using a best-first search strategy [44],
- *Naïve Bayes (NB)* algorithms that implement a simple form of Bayesian network that assumes that the predictive attributes are conditionally independent, and that no hidden or latent attributes influence the prediction process [66],
- *Decision Tables (DT)* based on a decision table format that may include multiple exact condition matches for a data item, computing the result by a majority vote [74],

- *Logistic Model Trees (LMT)* that combine linear logistic regression and tree induction [76],
- *Hidden Naïve Bayes (HNB)* algorithms that use the mutual information attribute weighted method to weight one-dependence estimators [141].

We discuss their experimental evaluation in Chapter 8.

5.3 Failure Prediction

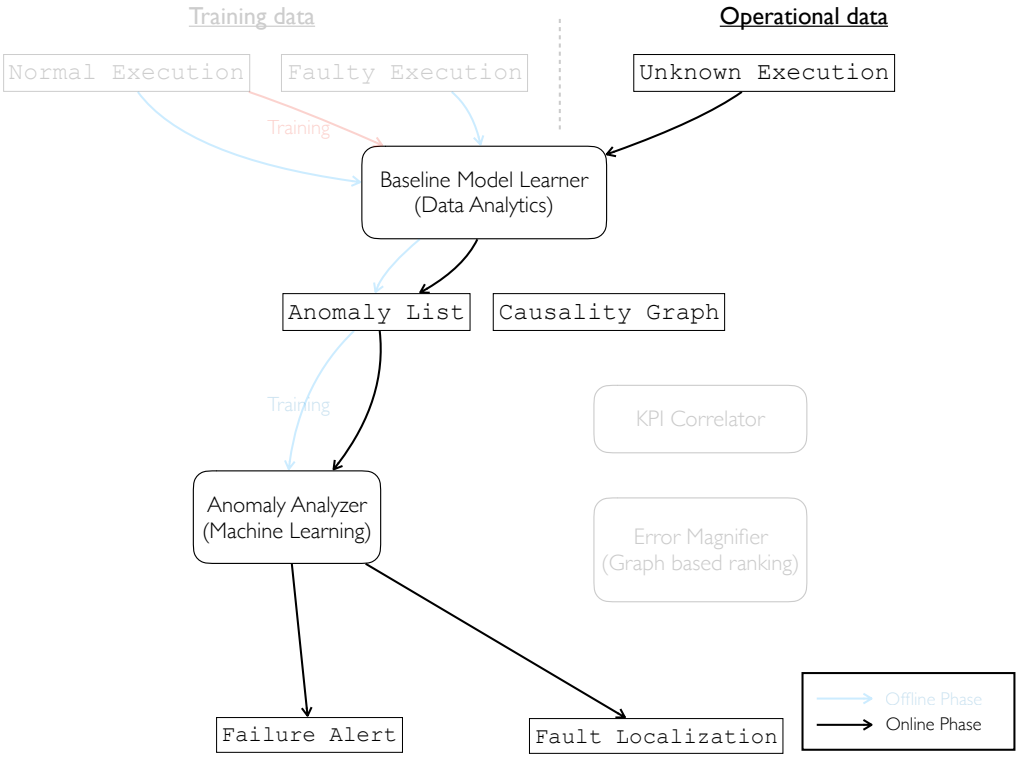

Figure 5.3. Workflow of Online Failure Prediction in PreMiSE

Figure 5.3 shows to the workflow of the Online Failure Prediction phase at operational time.

The Online Failure Prediction is composed of two elements, the *Baseline Model Learner* and the *Anomaly Analyzer* . At operational time, the *Baseline Model Learner* works on a model obtained from monitoring data under normal execution. the *Anomaly Analyzer* extracts the signature model from the Anomaly List produced by *Baseline Model Learner* under both normal and faulty executions.

The *Baseline Model Learner* determines the anomaly state of the KPIs, and encodes the information in a_i, thus at a given timestamp or time interval the *Anomaly Analyzer* leverages the signature model and tries to map the vector to a label it has seen before. If it finds a match to a (T_{fault}, R_{fault}) label, it generates a *Failure Alert* and a *Fault Localization*. A *Failure Alert* warns

the administrator that a potential failure is expected, while the *Fault Localization* indicates the fault type and location: T_{fault} and R_{fault}.

5.4 Summary

PreMiSE further processes the data produced by the *Baseline Model Learner* to derive failure prediction and fault localization. As *Baseline Model Learner* has reduced the data of each KPI from concrete data to binary data, the scale and dimension of the input for *PreMiSE* is appropriate to analyze.

PreMiSE works by mapping a specific set of anomalies to a failure prediction and fault localization. *PreMiSE* infer this mapping through machine learning techniques presented in Section 5.2. To prepare training data for the machine learning models, *PreMiSE* collects data under faulty executions with fault injection described in Section 5.1. With the trained model, *PreMiSE* can predict failure and localize faults online with the workflow covered by Section 5.3.

PreMiSE requires fault injection, that may fail to apply to specific systems or critical components that are stateful and is not resilient to data loss.

Chapter 6

LOUD

In this chapter, I present the design and implementation of the *LOUD* component that identifies anomalies by relying on training with normal data only, and is very useful when it is not possible to seed failures in the system for training purposes.

LOUD is composed of a *Signature Extraction* phase at training time and a *Failure Prediction* phase at operational time. The Signature Extraction phase corresponds to the *PreMiSE* Signature Extraction operating with data from normal executions only. The *Failure Prediction* localizes faults with a graph based algorithm that operates on the Causality Graph produced by the *Baseline Model Learner* .

Section 6.1 discusses how *LOUD* builds the prediction model in the presence of data coming from only normal execution. Section 6.2 presents the *LOUD* workflow for predicting failure and localizing faults.

6.1 Signature Model Extraction at Training Time

Figure 6.1 shows the workflow of the *LOUD* Signature Model Extraction at training time. The *LOUD* Signature Model Extraction works like the *PreMiSE* Signature Model Extraction on KPIs collected during normal execution only. The *LOUD* Signature Model Extraction produces an Anomaly List with the constant lable "Normal":

$$a_1, a_2, \ldots, a_n, \quad \underline{Normal}$$

LOUD identifies anomalies with a machine learning algorithm that finds outliers with training data on a single class.

One Class Support Vector Machine (OCSVM) is a typical model for one class training data. As Figure 6.2 shows, at training time, OCSVM builds a model by fitting a borderline to the training data. At runtime, OCSVM considers data within the borderline as normal, and daga outside the border line as anomalous.

When trained with a reasonable set of data, OCSVM can construct a borderline that well balance false positives and false negatives, that is, the model is not too strict to exclude too many normal instances, and not too loose to include too many abnormal instances. We discuss how we selected the window size and a set of events for a proper training in Chapter 7. *LOUD* uses the model built at training time to issue *Failure Alert*s.

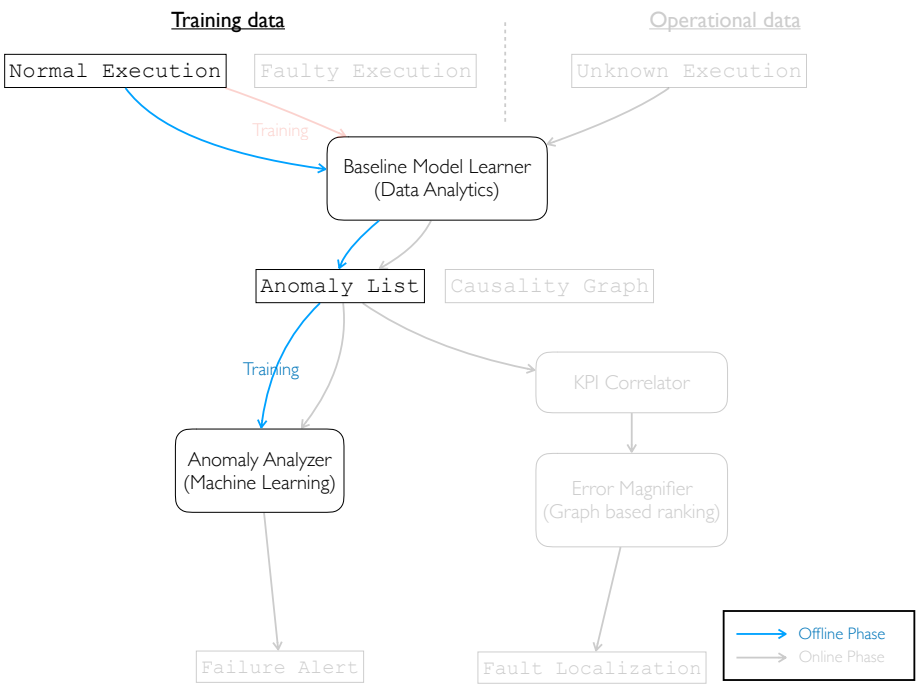

Figure 6.1. Workflow of Signature Model Extractor in LOUD

6.2 Online Failure Prediction

Figure 6.3 shows the *LOUD* online workflow that issues Failure Alerts at runtime. The *Anomaly Analyzer* issues *Failure Alerts* that indicate possible future failures, and is paired with the *KPI Correlator* and the *Error Magnifier* that locate the related faults. All our experiments immediately indicate that determining the faulty resource from the order of the appearance of anomalies over time is largely imprecise and leads to an unacceptable amount of false alarms. The *KPI Correlator* and the *Error Magnifier* precisely locate faults by using the *Causality Graph* that *Baseline Model Learner* produces at training time. The Causality Graph models the correlations among KPIs, which implicitly encode a model of anomaly infection chain. Below I describe how the *KPI Correlator* and the *Error Magnifier* locate faults b exploiting the *Anomaly List* and the *Causality Graph*.

6.2.1 KPI Correlator

In the Causality Graph nodes correspond to KPIs and weighted arcs to KPI correlations. Analyzing the complete Causality Graph with a node for each KPI is too time-consuming for large systems with an enormous amount of KPIs. We recall that KPIs are pairs $\langle metric, node \rangle$, and a large system may have several tens or hundreds of thousands of nodes and may collect hundreds or thousands of metrics for each node.

KPI Correlator reduces the portion of the causality graph to be analyzed by removing sub-components of the Causality Graph that are not relevant for the specific analysis, and produces

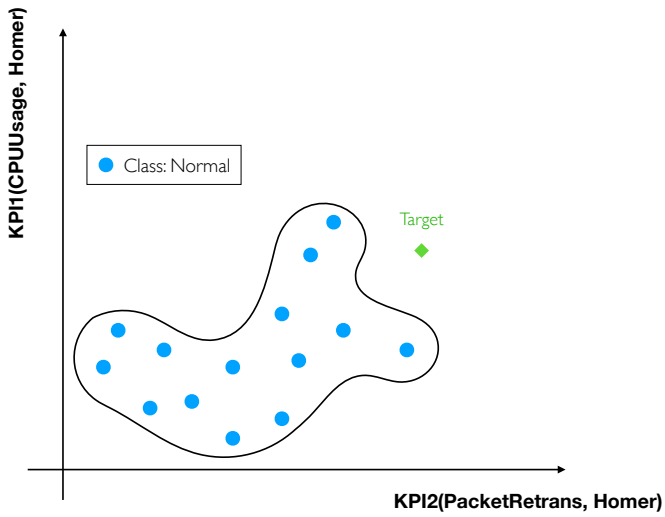

Figure 6.2. A sample One Class Support Vector Machine in LOUD

a **propagation graph**. Figure 6.4 shows a sample propagation graph (on the right-hand side) that has been generated from the causality graph (on the left-hand side) in our experiments. In the graph, thickness of the edges indicates the significance of correlations between metrics, which provides quantitative parameters for further analysis. The causality graph reported in the figure contains a small amount of nodes, and illustrate the reduction process, not the reduction rate that in the graph is limited (fraction of grey over red nodes), but may become extremely large when the size of the causality graph grows, as illustrated in details in the experiments reported in Section 8.

The *KPI Correlator* reduces the size of the causality graph by simply removing non anomalous KPIs:

Let the original graph be $G(V, E)$, and define a function $f_{anomalous}(v_i)$ which returns $true$ if vertex(KPI) v_i is anomalous otherwise $false$. We generate the *propagation graph* $G'(V', E')$ following two rules:

- $V' = \{v_i | v_i \in V \ and \ f_{anomalous}(v_i) = true\}$
- $E' = \{e_i(v_{i1}, v_{i2}) | e_i \in E \ and \ f_{anomalous}(v_{i1}) \vee f_{anomalous}(v_{i2}) = true\}$

6.2.2 Error Magnifier

The *Error Magnifier* analyzes the *propagation graph* to identify the location of the fault that may lead to the predicted failure. *Error Magnifier*'s analysis is based on the assumption that the causal relationship indicates the directed "influence" between the KPIs. *Error Magnifier* assumes that an anomalous KPI will likely affect its neighbors in the graph to be anomalous as well, depending on the strength of their causal relationship.

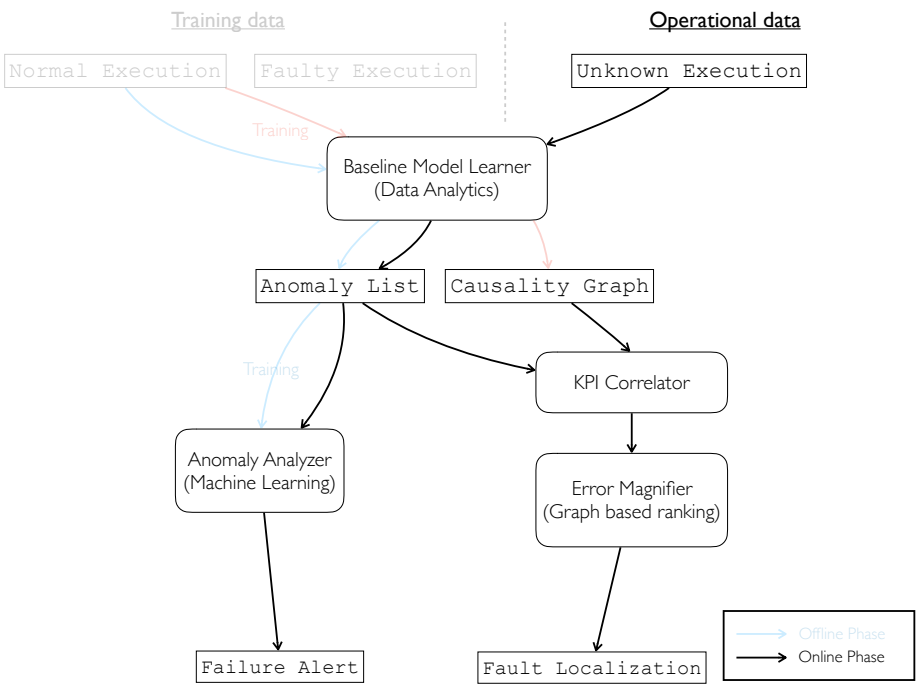

Figure 6.3. Workflow of Online Failure Prediction in LOUD

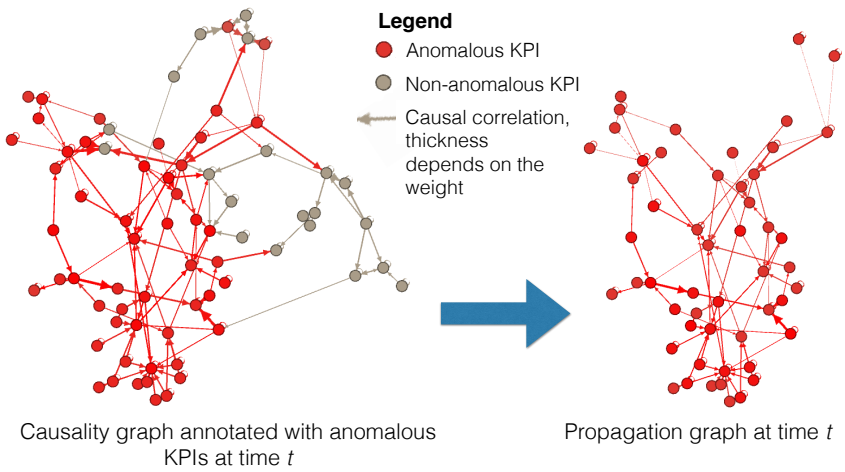

Figure 6.4. A sample causality graph and a corresponding propagation graph

As a simple example, in a client-server system, the client sends task to the server for processing. The more packets the client sends in a time interval, the more resource, such as CPU and memory, the server uses to process the received packets. Ideally the Causality Graph will capture the causality relation between the KPIs *{number of packets sent on the client}*, *{CPU usage on the server}* and *{memory usage on the server}*. If a fault in the client causes the client entering an

infinite loop that keeps sending trash requests to the server, the server will waste a lot of resources to process the request and all three KPIs will be anomalous. By reasoning the *propagation graph*, *Error Magnifier* is able to determine that the anomalies origins from the KPI *{number of packets sent on the client}*, and infer the client is the more likely location of the fault.

To identify the most relevant nodes in the *propagation graph*, *LOUD* computes graph centrality indicators [77, 83, 111] to infer the anomalous KPIs that most likely lead to a potential failure.

In graph theory and network analysis, there are different centrality indicators to infer the relevance of vertexes, that are computed with different algorithms that assign scores to nodes (KPIs). In the next sections I discuss the four algorithms that *LOUD* implements to compute the node ranking. Section 8 discusses the experimental comparison of the algorithms.

Eigenvector Centrality

The *eigenvector centrality* of a weighted graph is a vector $\mathbf{c}$ whose i^{th} component c_i represents the score of the node i [111]. In a graph with n nodes, the score c_i is defined as

$$c_i = \mu \sum_{j=1}^{n} W_{ij} c_j \tag{6.1}$$

that is, it is proportional to the sum of the weights W_{ij} of the edges that connect the node i to its neighbors, where μ is a proportionality constant that derives from the matrix of the eigenvalues. The eigenvector centrality score can be computed iteratively from Equation 6.1, and the solution is unique for a strongly connected graph with non-negative weights, as the case of propagation graphs.

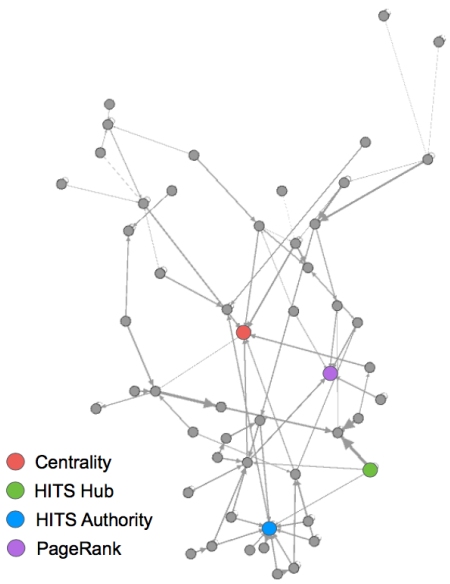

Figure 6.5. Sample rankings for the nodes of the propagation graph of Figure 6.4

The eigenvector centrality may produce misleading results when the graph contains very few nodes that act as strong hubs and create a “condensation” or “winner-takes-it-all” phe-

nomenon [83]. This is because the eigenvector centrality assigns to such nodes a much higher score than to most of the nodes, thus hiding the relevance of the other nodes which get a score close to zero, even if they might be related to the faulty resource. In our case, this phenomenon may potentially bias the localization and cause false positives.

Non-Backtracking Centrality

The *non-backtracking centrality* is a variation of the eigenvector centrality that produces more accurate centrality scores in the presence of a condensation effect. Intuitively, the non-backtracking centrality weights a node i as the sum of the weights of its neighbors computed in the absence of the node i itself [83].

The non-backtracking centrality can be computed from equation (6.1) by substituting the adjacency matrix with the non-backtracking matrix B, defined as $B_{ij} = W_{ij}$ *iff* (i, j) forms a non-backtracking path of length 2. A non-backtracking path is a directed path in which no edge is the inverse of the preceding edge [93].

The assumption behind the eigenvector and non-backtracking centrality is that a node is important if it is attached to many edges coming from many other important nodes. In the propagation graph, these nodes represent KPIs which are more influenced by the spread of anomalies.

HITS Algorithm for Hubs and Authorities

The Hyperlink Induced Topic Search (HITS) algorithm assigns a high score to nodes that are linked with other highly scored nodes [73]. This intuitively corresponds to identifying the KPIs that have a strong influence on the behavior of the system, and thus the corresponding resources. These relevant nodes might be of two kinds: *authorities*, which are the nodes that are the destination of edges from highly ranked nodes, and *hubs*, which are the nodes with many outgoing edges leading to highly ranked nodes.

The equations to score nodes as authorities (a_i) or hubs (h_i) are:

$$\sum_{j=1}^{n} W_{ij}\, a_j = \delta\, h_i, \qquad \sum_{k=1}^{n} W_{ik}\, h_k = \nu\, a_j$$

where δ and ν are proportionality constants related to eigenvalues of W. The HITS algorithm combines the two equations to compute iteratively a hub score and an authority score for each node in the graph.

PageRank

PageRank [77] scores nodes according to both the number of incoming edges and the probability of anomalies to randomly spread through the graph (*teleportation*). The PageRank score r_i of node i is computed iteratively from the score r_j of the nodes j that are sources of edges directed to i:

$$r_i = \sum_{j \in \mathcal{N}_{in}(i)} W_{ij}\, r_j + \frac{1-\alpha}{n} \tag{6.2}$$

where $\mathcal{N}_{in}(i)$ is the set of nodes j that are sources of edges directed to i, and α is the teleportation probability (*LOUD* uses $\alpha = 0.85$). The interested reader can refer to the excellent description of Langville et al. [77] for a detailed survey on the HITS algorithm and PageRank.

Pagerank implements score teleportation between nodes. *DyFAULT* anomaly detector produces anomalies from time series of data, and the resulting temporal evolution of the anomaly set reflects the underlying propagation. In the model, causality significance is a proper approximation of teleport probability between nodes, and as such helps us localize the key anomalies contributing to the propagation graph.

Figure 6.5 exemplifies the effects of the different centrality indices on the propagation graph of Figure 6.4. The Figure shows in different colors the nodes with the highest score for each algorithm. The red node is the top ranked node of both eigenvector and non-backtracking centrality, which means that the centrality index is evenly distributed through many important nodes in the graph, and the variations of the non-backtracking centrality have a low impact on the scores. The Figure illustrates the dual role of hubs and authorities: there is a directed edge from the top HITS Hub node to the top HITS Authority node. The role of the purple highest PageRank node is similar to the role of the red highest centrality node, since both are destination of highly weighted edges. The two nodes are different due to the impact of the PageRank teleportation.

LOUD computes the four centrality indices with the power iteration algorithm that computes the indices with an increasingly better approximation at each iteration, and can thus be interrupted and later resumed at any time to improve the precision of the localization.

LOUD identifies the likely faulty resource as the most frequent resource in the top 20 KPIs according to the selected centrality index, and does not suggest a fault location in the absence of a most frequent resource in the set. The intuition is that a faulty resource is likely to show an anomalous behavior for multiple KPIs, which in turn are likely to affect several other resources, and thus their KPIs, of the system. As a consequence a faulty resource is likely to be present with multiple KPIs in the top part of the ranking.

6.3 Summary

Comparing to *PreMiSE* , the offline phase of *LOUD* does not need to engage fault injection to sample data under abnormal states. As presented in Section 6.1, *LOUD* employs one class machine learning techniques to learn from the anomalies under normal execution, and derives a model which is able to tell whether a given set is normal or not, treating a positive result as a failure prediction.

As Section 6.2 covered, since one class machine learning models can only provide a binary value as the failure prediction result, *LOUD* uses two subcomponents, namely *KPI Correlator* and *Error Magnifier*, with graph ranking algorithms, to infer the suspicious faulty location.

Execution type of the target system limits the input data for *LOUD* to build the model, which risks in the decrease of its accuracy in predict failures and localize faults, but provides wider applicability to adapts to different system requiremens. Administrator can choose between *PreMiSE* and *LOUD* as the target systems workflow by making a trade-off.

Chapter 7

Experimental Infrastructure

In this chapter, I present the experimental methodology that I define to evaluate *DyFAULT*. I discuss the testbed configuration, the fault seeding framework, the research questions and the evaluation measurements.

Section 7.1 describes the testbed configuration. Section 7.2 discusses the selection of the types of faults seeded in the experiments. Section 7.3 presents the KPIs we sample from the system and the *Baseline Model Learner* implementation used in the experiments. Section 7.4 presents the research questions that I addressed in the experiments. Section 7.5 describes the metrics that I use for evaluating *DyFAULT*.

7.1 Testbed Configuration

We conducted our experiments on data collected on an IP Multimedia Subsystem (IMS) service running on a physically distributed multi-tier cloud system, an IMS that support IP phone calls. We executed the system with a profile of a real world phone call traffic that we adapted to fit our system capacity, resulting in more than 150 online simultaneous communications, generating traffic of up to 13,000 queries per minutes.

7.1.1 Hardware Configuration

To easily set up metrics collection from low level physical machines, I build the testbed distributed system from scratch. Restricted to the difference between a research lab and the real world industrial environment, the scale of the system I use for the experiments can not be ideally big. I try to maximize the complexity of the system by using a number of desktop machines instead of few high perforce servers. In this way, single point problem will be more likely and connections between machines will also introduce uncertainty.

Table 7.1 shows the hardware configuration of the machines that I use in the experiments. There are three kinds of machines, depending on the roles that the machines play in the system: (i) a controller node responsible for running the management services necessary for the virtualization infrastructure, (ii) six compute nodes that run VM instances, (iii) a network node responsible for network communication among virtual machines.

I deploy the prototype of *DyFAULT* on a standalone machine, including the KPI monitoring data integration, the *Baseline Model Learner* , the *PreMiSE* and *LOUD* component. The machine is

Table 7.1. Hardware configuration

Host	Controller	Network	Compute (x2)	Compute (x4)
CPU	*Intel(R) Core(TM)2 Quad CPU Q9650 (12M Cache, 3.00 GHz, 1333 MHz FSB)*			
RAM	4 GB	4 GB	8 GB	4 GB
Disk	250 GB SATA hard disk			
NIC	Intel(R) 82545EM Gigabit Ethernet Controller			

a Red Hat Enterprise Linux Server release 6.3 with Intel(R) Core (TM) 2 Quad Q9650 processor at 3GHz frequency and 16GB RAM.

Since the machine that runs *DyFAULT* services is not part of the target system, *DyFAULT* has negligible impact on the evaluation result.

7.1.2 Infrastructure Configuration

I install Ubuntu 14.04 LTS on the 8 machines as the operating system. I deploy OpenStack [124] upon the operating system. OpenStack is an open-source implementation of cloud computing architecture, featuring an Infrastructure-as-a-Service (IaaS) framework. I installed OpenStack version Icehouse with KVM [122] as hypervisor.

At the application layer, I deployed Clearwater [100] on the cloud-based infrastructure. Clearwater is a standard open source solution adopted by several large telecommunication companies for their IP-based voice, video and message services. Clearwater is specifically designed to massively scale on a cloud-based infrastructure, and is a product originated from the current trend of migrating traditional network functions from inflexible and expensive hardware appliances to cloud-based software solutions. Our Clearwater deployment consists of the following virtual machines:

Bono: the entry point for all client communication in the Clearwater system.

Sprout: the handler of client authentications and registrations.

Homestead: a Web service interface to Sprout for retrieving authentication credentials and user profile information.

Homer: a standard XML Document Management Server that stores MMTEL (MultiMedia Telephony) service settings for each user.

Ralf: a service for offline billing capabilities.

Ellis: a service for self sign-up, password management, line management and control of multimedia telephony service settings.

Each component runs on a different VM. Each VM is configured with 2 vCPU, 2GB of RAM and 20GB hard disk space, and runs the Ubuntu 12.04.5 LTS operating system. Our multi-tier distributed system is thus composed of eight machines running components from three tiers: the operating system, infrastructure and application tiers, running Linux, OpenStack, and virtual machines with Clearwater, respectively. Figure 7.1 shows the architecture of the testbed.

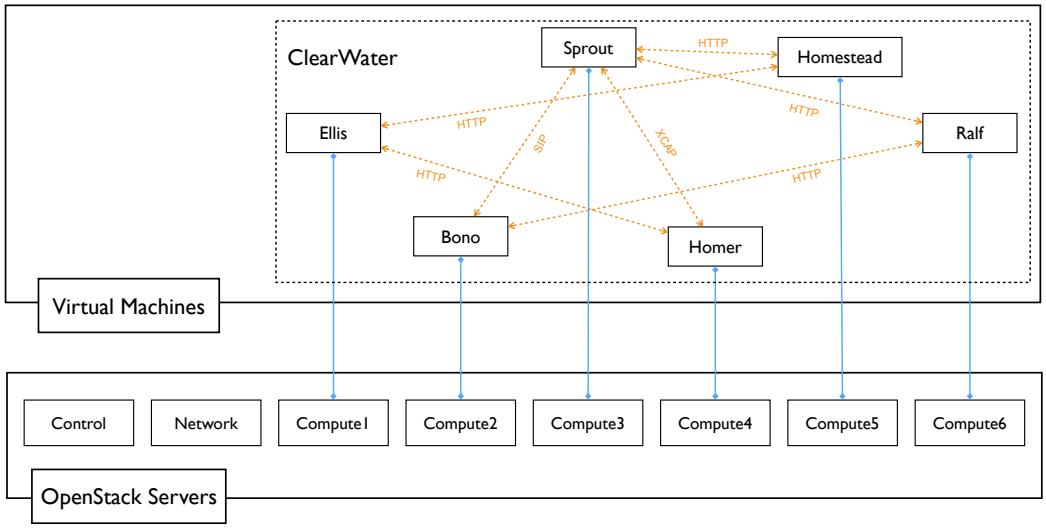

Figure 7.1. Reference Logical Architecture

7.1.3 Workload shaping

I define the workload used in the experimental evaluation to replicate the shape of real SIP traffic as experienced by our industrial partners in the telecommunication domain. I carefully tuned the peak of the workflow to use as much as 80% of CPU and memory. I generate the SIP traffic with the SIPp traffic generator [103], which is an open source initiative from Hewlett-Packard (HP) and is the defacto standard for SIP performance benchmarking.

SIPp can simulate the generation of multiple calls using a single machine. The generated calls follow user-defined scenarios that include the exact definition of both the SIP dialog and the structure of the individual SIP messages. In the evaluation, I use the main SIP call flow dialogs as documented in Clearwater[1].

The workload includes a certain degree of randomness, and generates new calls based on a call rate that changes according to calendar patterns. In particular, I consider two workload patterns:

Daily variations The system is busier on certain days of the week. In particular, I consider a higher traffic in working days (Monday through Friday) and lower traffic in the weekend days (Saturday and Sunday). Figure 7.2 graphically illustrates the structure of our workload over a period of a week.

Hourly variations To resemble daily usage patterns, our workload is light in the night and heavy in the day with two peaks at 9am and 7pm, as graphically illustrated in Figure 7.3.

[1] http://www.projectclearwater.org/technical/call-flows/

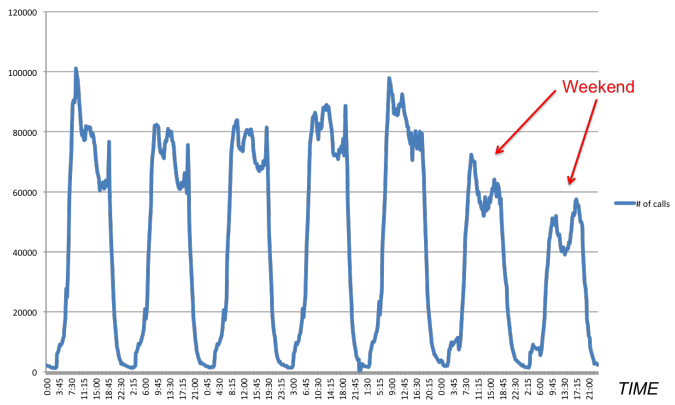

Figure 7.2. Plot with calls per second generated by our workload over a week

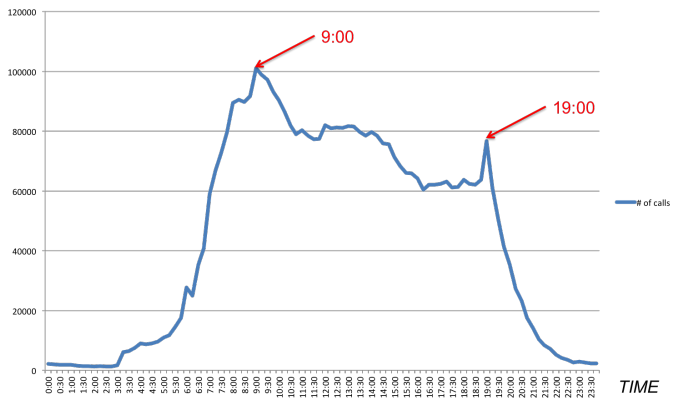

Figure 7.3. Plot with calls per second generated by our workload over a day

7.2 Fault Seeding

Fault seeding consists of introducing faults in a system to reproduce the effects of real faults, and is a common approach to evaluate the dependability of systems and study the effectiveness of fault-tolerance mechanisms [8, 113] in test or production environments [9, 109].

Since I use a cloud-based system to evaluate *DyFAULT*, I identify a set of faults that are common problems in cloud-based applications. I analyze a set of issue reports[2] of some relevant cloud projects to determine the most relevant fault types that threat Cloud applications. I analyze a total of 106 issue reports, 18 about KVM[3], 62 about OpenStack[4], 19 about CloudFoundry[5], and 7 about Amazon[6], and I informally assess the results with our industrial partners that operate in the telecommunication infrastructure domain.

I classify the analyzed faults in thirteen main categories. Figure 7.4 plots the percentage of faults per category in decreasing order of occurrence for the analyzed fault repositories. The figure indicates a gap between the three more frequent and the other categories of faults. I experimented with the three most frequent categories: *Network*, *Resource leaks* and *High overhead* faults. *Network* faults consist of networking issues that typically affect the network and transport layers, such as packet loss problems. *Resource leaks* occur when resources that should be available for executing the system are not obtainable, for instance because a faulty process does not release memory when not needed anymore. *High overhead* faults occur when a system component cannot meet its overall objectives due to inadequate performance, for instance because of poorly implemented APIs or resource-intensive activities.

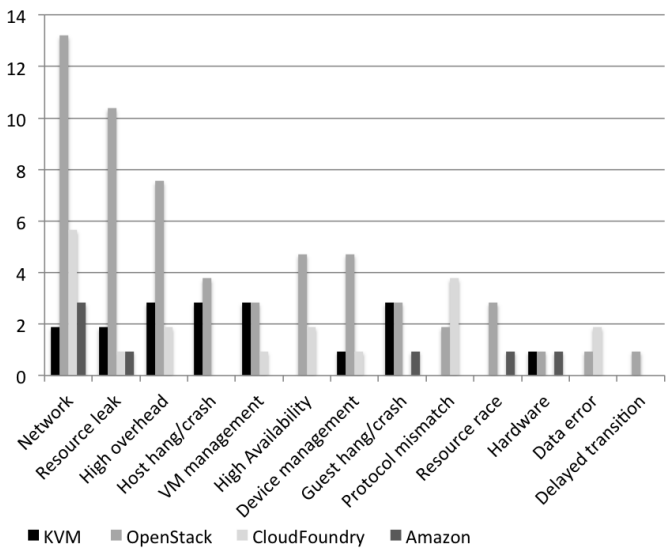

Figure 7.4. Occurrences of categories of faults in the analyzed repositories

Based on the results of this analysis, I evaluate *DyFAULT* with injected faults of six types, that

[2] I conducted the analysis in July 2014 selecting the most recent issue reports at the time of the inspection.
[3] https://bugzilla.kernel.org/buglist.cgi?component=kvm
[4] https://bugs.launchpad.net/openstack
[5] https://www.pivotaltracker.com/n/projects/956238
[6] http://aws.amazon.com

characterize the three top ranked categories of faults in Figure 7.4 and hence provides a wide coverage of typical faults in cloud-based systems: *Network faults* that depend on *Packet Loss* due to *Hardware* and *Excessive workload* conditions, increased *Packet Latency* due to network delay and *Packet corruption* due to errors in packet transmission and reception, *Resource leak faults* that depend on *Memory Leaks*, and *High overhead faults* that depend on *CPU Hogs*. In details,

i.- A packet loss due to hardware conditions drops a fraction of the network packets, and simulates the degradation of the cloud network;

ii.- A packet loss due to excessive workload conditions corresponds to an extremely intensive workload, and causes an intensive packet loss;

iii.- An increased packet latency leads to long latency in packet transmission;

iv.- A corruption due to channel noise, routing anomalies or path failures, simulates degraded packet delivery performances;

v.- A memory leak fault periodically allocates some memory without releasing it, simulates a common software bug, which severely threaten the dependability of cloud systems;

vi.- A CPU Hog fault executes some CPU intensive processes that consume most of the CPU time and cause poor system performance.

I limited the investigation to the most relevant categories of faults to control the size of the experiment, which already involves an extremely large number of executions. The results that I discuss in Chapter 8 indicate steadily effectiveness of *DyFAULT* across all the faults considered in the experiments. I expect comparable results for other fault categories with the same characteristics of the considered ones, namely faults that lead to the degradation of some KPI values over time before a failure. This is the case of most of the fault categories of Figure 7.4, with the exception of host and guest crashes, which may sometime occur suddenly and without an observable degradation of KPI values over time. Confirming this hypothesis and thus extending the results to a broader range of fault categories would require additional experiments.

I inject packet loss, packet latency, packet corruption, memory leaks and CPU Hogs faults into both the host (Openstack) and guest (Clearwater) layers, and excessive workload faults by altering the nature of the workload, following the approaches proposed in previous studies on dependability of cloud systems [8] and on problem determination in dynamic clouds [113].

I study a wide range of situations by injecting faults according to three activation patterns:

Constant: the fault is triggered with a same frequency over time.

Exponential: the fault is activated with a frequency that increases exponentially, resulting in a shorter time to failure.

Random: the fault is activated randomly over time.

Overall, I seeded 12 faults in different hosts and VMs. Each fault is characterized by a fault type and an activation pattern.

7.3 KPI collection and the Baseline Model Learner

Our prototype implementation of the monitoring infrastructure collects a total of 633 KPIs of 96 different types from the 14 physical and virtual machines that comprise the prototype. I collected KPIs at three levels: 162 KPIs at the application level with the SNMPv2c (Simple Network Management Protocol) monitoring service for Clearwater [22], 121 KPIs at the IaaS level with the OpenStack telemetry service (Ceilometer) for OpenStack [123] and 350 KPIs at the operating system level with a Linux OS agent that I implemented for Ubuntu. I selected the KPIs referring to the multi-tired distributed nature of our prototype and the low-impact requirements that characterize most industrial scale systems. I collected KPIs from all the tires characterizing the system, by relying on already available services, when possible, and on ad-hoc build monitors otherwise. I collected only KPIs that can be monitored with no functional impact and negligible performance overhead. As expected, *DyFAULT* did not impose any limitation on the set of collected and processed KPIs, and I expect this to be valid in general.

At the application tier, *DyFAULT* collects both standard SNMP KPIs, such as communication latency between virtual machines, and Clearwater specific KPIs, such as the number of rejected IP-voice calls. At the IaaS tier, *DyFAULT* collects KPIs about the cloud resource usage, such as the rate of read and write operations executed by OpenStack. At the operating system tier, *DyFAULT* collects KPIs about the usage of the physical resources, such as consumption of computational, storage, and network resources. In our evaluation, I used a sampling rate of 60 seconds.

DyFAULT elaborates KPIs from both simple and aggregated metrics, that is, metrics that can be sampled directly, such as CPU usage, and metrics derived from multiple simple metrics, for example the call success rate, which can be derived from the number of total and successful calls, respectively.

I implemented the *Baseline Model Learner* on release 1.3 of *IBM ITOA-PI* [63], a state-of-the-art tool that computes the correlation between pairs of KPIs. *IBM ITOA-PI* detects anomalies by implementing the following anomaly detection criteria: normal baseline variance, normal-to-flat variation, variance reduction, Granger causality, unexpected level, out-of-range values, rare values[7], and issues alarms after revealing anomalies in few consecutive samples. *IBM ITOA-PI* can analyze a large volume of data in real-time (the official IBM documentation indicates up to $500,000$ KPIs per server)[8], and provides both model learning and online checking capabilities. The learning phase requires collecting correct executions for a wide time interval that includes an accurate sample of normal operation behaviors. I used a learning time interval of two weeks.

The *Anomaly Analyzer* includes different classifiers for signature extraction and online prediction. For *PreMiSE* I implemented the *Anomaly Analyzer* on top of the Weka library [127], a widely used open-source library that supports several classic machine learning algorithms. For *LOUD*, I adopted the OneClassSVM implementation from the Python Scikit-learn library [97].

7.4 Research Questions

I identify three categories of research questions to evaluate *DyFAULT*: *Effectiveness*, *Comparison* and *Overload*. *Effectiveness* refer to the effectiveness of *PreMiSE* and *LOUD* algorithms. *Compari-*

[7] https://www.ibm.com/support/knowledgecenter/SSJQQ3_1.3.6/com.ibm.scapi.doc/intro/r_oapi_adminguide_algorithms.html

[8] https://www.ibm.com/support/knowledgecenter/en/SSJQQ3_1.3.4/com.ibm.scapi.doc/intro/c_oapi_intro_scalability.html

son refer to a comparative evaluation of *DyFAULT* with respect to state-of-the-art techniques. I compare our *DyFAULT* with *IBM ITOA-PI* a state-of-the-art commercial tool for failure prediction. *Overhead* refer on the costs the approach.

Effectiveness

RQ1 *Does the size of the sliding window impact on the effectiveness of* PreMiSE *and* LOUD*?*
As common practice [94] suggests a sliding window help generate prediction on recent observations, I experimented with different sliding window in the execution phase of *PreMiSE* and the prediction phase of *LOUD*, and compared the effectiveness of them to show the impact of aggregation window. We tune the fault activation to reach the maximum frequency in 2 to 3 hours depending on the activation pattern. From practice, in the first 30 minutes the frequency is not significant enough to reveal symptoms and metric deviation, we therefor select three sliding windows of 30 minutes, 60 minutes and 90 minutes to aggregate anomalies.

RQ2 *Does the choice of algorithm change the effectiveness of* PreMiSE *prediction?*
I executed *PreMiSE* with different algorithms in the presence of different types of faults, and evaluate its effectiveness in failure prediction.

RQ3 *Does the choice of centrality index impact on the precision of* LOUD *fault localization?*
As discussed in Chapter 6, *LOUD* implements five centrality-based indices that are potentially suitable for fault localization. I experimentally compared the effectiveness of the different indices, to evaluate their impact on fault localization.

RQ4 *Do the type of the faulty resource, the type of the fault and the activation pattern impact on the effectiveness of* LOUD *localization?*
I evaluated the accuracy of *LOUD* with different kinds of faults seeded in various types of resources with distinct activation patterns to evaluate the impact of these factors on *LOUD* precision. I executed the experiments with the PageRank centrality index, which the experiments for RQ1 indicate as the most effective index for the *LOUD* localization.

RQ5 *How early can* PreMiSE *and* LOUD *predict a failure?*
I measured the important timestamps, such as time in the prediction, and the time when the system crashes without error fixing, to access how quick can *PreMiSE* and *LOUD* predict a failure.

Comparison

Commercial solutions for predicting failures and locating faults are extremely expensive and rely on techniques hidden to the general public; Academic prototypes are seldom available and installable for large scale experimentation. After considering different alternatives, I decided to compare *DyFAULT* with *IBM ITOA-PI*, a state-of-the-art solution to detect early symptoms of failures, largely used in commercial environments to predict failures[9], and easily configurable for cloud infrastructures[10].

I compared *PreMiSE* and *LOUD* with *IBM ITOA-PI* [63] to investigate the following question:

[9] https://developer.ibm.com/itom/docs/predictive-insights/
[10] https://www.ibm.com/support/knowledgecenter/SSMKFH/com.ibm.apmaas.doc/install/integ_predictviveinsights_config.html

RQ6 *Can* PreMiSE *and* LOUD *predict failures more accurately than* IBM ITOA-PI *?*
I executed the three approaches on the same set of experiments to compare the accuracy of the approaches.

Overhead

I investigated the impact of *DyFAULT* on the overall performance of a cloud-based system by addressing the following research question:

RQ7 *What is the overhead of* DyFAULT *on the performance of the target system?* This question is particularly relevant in the context of multi-tier distributed systems with strict performance requirements. Thus, designed an experiment referring to such applications. *DyFAULT* executes the resource-intensive tasks, that is, anomaly detection, failure prediction, and fault localization, on a dedicated physical server, and thus the overhead on the system derives only from monitoring the KPIs.

I evaluated the impact of monitoring the KPIs on the system performance by measuring the consumption of the different resources when running the system with and without KPI monitoring active. Both *PreMiSE* and *LOUD* solutions monitor the same KPIs, and thus share the same performance overhead.

7.5 Evaluation Measures

I addressed the research questions RQ1, RQ2 and RQ6 by using 10-fold cross-validation [134]. *DyFAULT* analyzes time series data, and collected anomalous KPIs every 5 minutes, to comply with the requirements of IBM *IBM ITOA-PI* [63], the time series analyzer used in the Baseline Model Learner of *DyFAULT*. I collected the *samples* necessary to apply 10-fold cross validation during the execution of a workload with sliding windows of length l. Since each run lasts 120 minutes and the size of the interval in the sliding window is 5 minutes, each workload execution produces $(120-l)/5$ samples that can be used for prediction. In the evaluation, I first studied the impact of l on the results (RQ1), and then used the best value in our contest for the other experiments.

Overall I collected samples from a total of 648 runs, which include 24 passing executions and 24 failing executions for each type of failure. A failing execution is characterized by a fault of a given type injected in a given resource with a given activation pattern. As discussed in Section 7.2, I injected faults of six different types (packet loss, excessive workload, packet latency, packet corruption, memory leak and cpu hog) following three activation patterns (constant, exponential and random). For all but excessive workload, I injected faults on five different target resources (the Bono, Sprout, and Homestead virtual machines in Clearwater and two compute nodes in OpenStack), resulting in $5 \times 3 \times 5 = 75$ failure cases. For excessive workload, I injected faults with three patterns with no specific target resource, since excessive workload faults target the system and not a specific resource. I thus obtained $75 + 3 = 78$ failure cases. To avoid biases due to the fault injection pattern, I repeated every experiment 8 times, thus obtaining 624 failing executions for the evaluation. The extensive investigation of the different fault types, activation patterns, and affected resources made the set of executions available for the experiment unbalanced between passing executions (24 cases) and failing executions (624 cases).

Since I use $l = 90$ for RQ2 and RQ6, I obtained a total of 4,782 samples collected from both passing and failing executions. The number of samples available for RQ1 is higher because I tried different values for l. To apply 10-fold cross-validation, I split the set of samples into 10 sets of equals size, using nine of them to learn the prediction model and the remaining set to compute the quality of the model. The *DyFAULT* failure prediction algorithm does not consider the order of the samples in time, since it classifies each sample independently from the others.

I evaluated the quality of a prediction model using the standard measures that are used to define contingency tables and that cover the four possible outcomes of failure prediction (see Table 7.2). I also measured the following derived metrics:

Precision: the ratio of correctly predicted failures over all predicted failures. This measure can be used to assess the rate of false alarms, and thus the rate of unnecessary reactions that might be triggered by the failure predictor.

Recall: the ratio of correctly predicted failures over actual failures. This measure can be used to assess the percentage of failures that can be predicted with the failure predictor.

F-Measure: the uniformly weighted harmonic mean of precision and recall. This measure captures with a single number the trade-off between precision and recall.

Accuracy: the ratio of correct predictions over the total number of predictions. The accuracy provides a quantitative measure of the capability to predict both failures and correct executions.

FPR (False Positive Rate): the ratio of incorrectly predicted failures to the number of all correct executions. The FPR provides a measure of the false warning frequency.

We observe that accuracy is an important metric in the presence of balanced data, but misses relevance in the presence of unbalanced data, which is often the case in our experiments that investigate abnormal cases. Thus we mainly focus on precision, recall and FPR, and keep accuracy for reference. Table 7.3 summarizes the derived metrics that I used by presenting their mathematical formulas and meanings.

Table 7.2. Contingency table

		Predicted	
		Failure	Not-Failure
Actual	Failure	True Positive (TP) *(correct warning)*	False Negative (FN) *(missed warning)*
	Not-failure	False Positive (FP) *(false warning)*	True Negative (TN) *(correct no-warning)*

As monitored data on hosts and guests are highly collerated that may interfere the results, I select a subset of the experiments to answer RQ3 and RQ4, by focusing on some typical faults on guest level machines. In total, I use 108 experiments as the study cases: 3 types of faults (packet loss, memory leak, CPU hog) injected in 3 different resources (Bono, Sprout, Homestead) with 3 different activation patterns repeated four times.

I measure the effectiveness of *LOUD* by computing precision, recall, and F-measure of the localization at each timestamp after the activation of the fault. In particular, I define true/false positive/negatives as follows:

Table 7.3. Selected metrics obtained from the contingency table

Metric	Formula	Meaning
Precision	$\frac{TP}{(TP+FP)}$	How many predicted failures are actual failures?
Recall	$\frac{TP}{(TP+FN)}$	How many actual failures are correctly predicted as failures?
F-measure	$2 * \frac{(Precision*Recall)}{(Precision+Recall)}$	Harmonic mean of $Precision$ and $Recall$
Accuracy	$\frac{(TP+TN)}{(TP+TN+FP+FN)}$	How many predictions are correct?
FPR	$\frac{(FP)}{(TN+FP)}$	How many correct executions are predicted as failures?

- *true positive (TP) at time t:* the *LOUD* localization produced at time t is correct,
- *false positive (FP) at time t:* the *LOUD* localization produced at time t is wrong,
- *false negative (FN) at time t: LOUD* cannot identify a most frequent resource in the top 20 anomalies, and thus does not produce a localization at time t.

I compute the precision as the rate of correct localizations out of the total set of localizations produced at time t: $\frac{TP}{TP+FP}$. I compute the recall as the rate of correct localizations out of the total set of localizations that should have been generated at time t: $\frac{TP}{TP+FN}$. I compute the F-measure as the harmonic average between precision and recall: $2\frac{precision*recall}{precision+recall}$.

I addressed the research question RQ5 by computing *the time needed to generate a prediction* and *the time between a failure prediction and its occurrence* from a total of 18 faulty runs lasting up to twelve hours. The former time measures the capability of *PreMiSE* to identify and report erroneous behaviors. The latter time estimates how early *PreMiSE* can predict failure occurrences. As shown in Figure 7.5, I define four specific measures:

Time-To-General-Prediction (*TTGP*): the distance between the time a fault is active for the first time and the time *PreMiSE* produces a general prediction,

Time-To-Failure-Specific-Prediction (*TTFSP*): the distance between the time a fault is active for the first time and the time *PreMiSE* predicts a specific failure type,

Time-To-Failure for General Prediction (*TTF(GP)*): the distance between the time *PreMiSE* predicts a general failure and the time the failure happens,

Time-To-Failure for Failure-Specific Prediction (*TTF(FSP)*): the distance between the time *PreMiSE* predicts a specific failure type and the time the system fails.

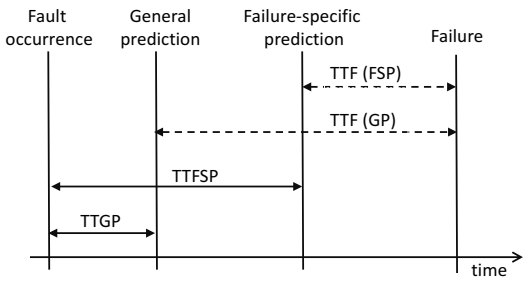

Figure 7.5. Prediction time measures

Fault occurrence is the time the seeded fault becomes active in the system,

General prediction is the first time *PreMiSE* signals the presence of a failure without indicating the fault yet, that is, it identifies an anomaly with an empty fault and resource,

Failure-specific prediction is the first time *PreMiSE* indicates also the fault type and the faulty resource,

Failure is the time the delivered service deviated from the system function. Failures depend on the seeded faults. In our case, failures manifest either as system crashes or as success rate dropping below 60%, as indicated in Chapter 8 when discussing RQ5.

To answer RQ7, I measured the resource consumption as (i) the percentage of *CPU* used by the monitoring activities, (ii) the amount of *memory* used by the monitoring activities, (iii) the amount of *bytes read/written* per second by the monitoring activities, and (iv) the packets received/sent per second over the *network* interfaces by the monitoring activities.

7.6 Summary

Though there are some other techniques aiming at predicting failures and localizing faults, replicating the same environment and the same self healing tools to compare them with *DyFAULT* is not trivial, due to the fact that some of them have specific setup and some of the tools are not open-source. We pick up *IBM ITOA-PI* as a baseline, since it is a commercial tool available and claims itself to be applicable for failure prediction in commercial cloud system.

Research questions mainly focus on effectiveness of *DyFAULT*, the comaprison of *DyFAULT* and *IBM ITOA-PI*, and the overhead that *DyFAULT* introduces to the target system.

Chapter 8

Experimental Results

In this chapter I present the results of the experiments that I executed to answer the research questions introduced in Section 7.4. I conclude the section with a discussion of the threats to the validity of the results.

RQ1: Sliding window size

DyFAULT builds the prediction models and analyzes anomalies referring to time sliding windows of fixed size. The sliding windows should be big enough to contain a sufficient amount of information to predict failures and small enough to be practical and sensitive to failure symptoms.

With this first set of experiments, I investigate the impact of the window size on the effectiveness of the prediction. I experimented with the seven algorithms described in Chapter 5 and one algorithm described in Chapter 6, each with sliding windows of size 60, 90 and 120 minutes to study the impact of the window size, and to choose the size for the next experiments. I built a total of 27 prediction models. I executed the prototype tool with the different prediction models both with and without seeded faults, for a total of 24 execution for 27 configurations, 26 of which corresponding to configurations each seeded with a different fault, and one no-faulty configuration, for a total of 648 executions. The configurations correspond to the raw of Table 8.2 that I discuss later in this section.

Figure 8.1 and Figure 8.2 compare the average precision, recall, F-measure and accuracy over all the experiments for both *PreMiSE* and *LOUD*. These results indicate that the window size has a moderate impact on the predictions, and that a window size of 90 minutes reaches the best prediction effectiveness among the experimented sizes.

Figure 8.3 and Figure 8.4 show the average false positive rates of both workflow for the different window sizes, and confirms the choice of a window of 90 minutes as the optimal choice among the evaluated sizes. The results collected for the individual algorithms are consistent with the average ones. In all the remaining experiments, I use 90-minutes windows.

RQ2: Predicting Failures and locating faults

I evaluated the effectiveness of *PreMiSE* as the ability of predicting incoming failures and identifying the kind and location of the related faults.

Table 8.1 shows precision, recall, F-measure, accuracy and False Positive Rate (FPR) of failure prediction and fault localization for *PreMiSE* with the prediction algorithms presented in

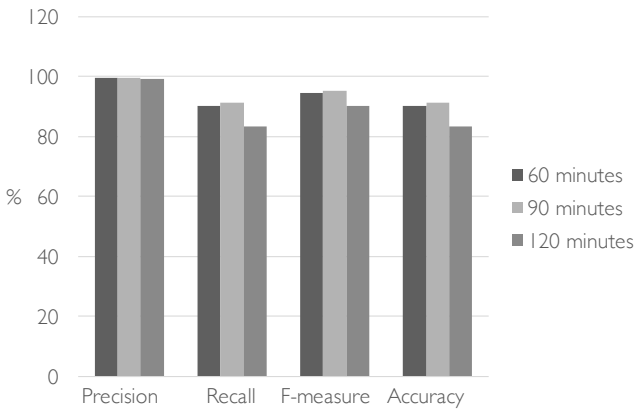

Figure 8.1. Average effectiveness of failure prediction approaches from PreMiSE with different sliding window sizes

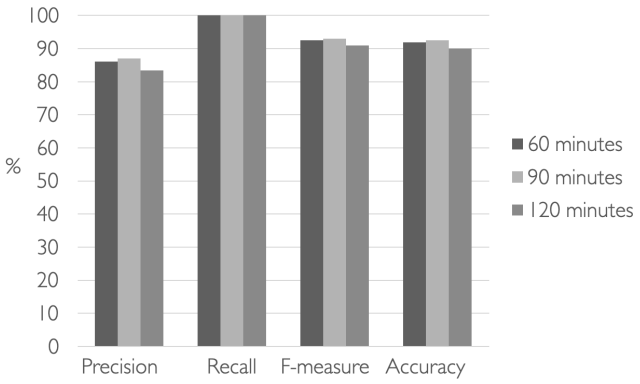

Figure 8.2. Effectiveness of failure prediction from LOUD with different sliding window sizes

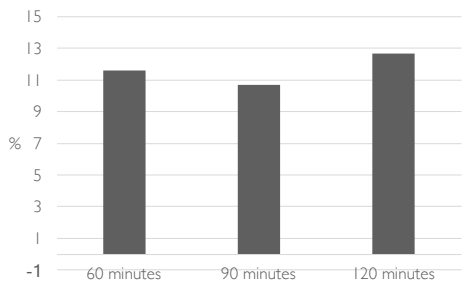

Figure 8.3. Average false positive rate from PreMiSE with different sliding window sizes

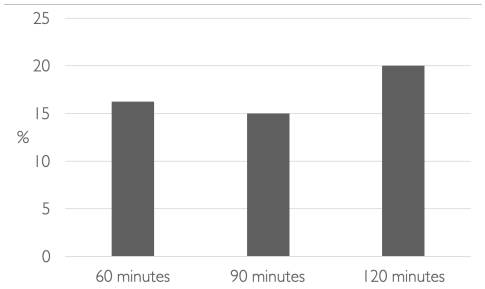

Figure 8.4. False positive rate from LOUD with different sliding window sizes

Table 8.1. Comparative evaluation of the effectiveness of PreMiSE prediction and localization with the different algorithms for generating signatures

Model	*Precision*	*Recall*	*F-measure*	*Accuracy*	*FPR*
BN	98.792	82.773	90.076	82.438	26.257
BFDT	100	97.631	98.801	97.719	0
NB	97.793	82.041	89.227	80.925	48.045
SVM	100	98.579	99.284	98.632	0
DT	99.976	88.135	93.683	88.555	0.559
LMT	100	98.751	99.372	98.797	0
HNB	100	91.516	95.570	91.831	0

Chapter 7. The table indicates that *PreMiSE* performs well with all the algorithms, with slightly better indicators for *LMTs (Logistic Model Trees)* that I select for the remaining experiments.

Table 8.2 shows the effectiveness of *PreMiSE* with LMT for the different fault types and locations. The metrics were calculated on a window basis as you need to make a forecast about each window. This means that windows that belong to both failed and correct executions are taken into account. *PreMiSE* suffered from only 74 false predictions out of 4,782 window samples. *PreMiSE* can quickly complete the offline training phase. To learn the *baseline model*, the data collected from two weeks of execution required less than 90 minutes of processing time. When the training phase runs in parallel to the data collection process, it completes almost immediately after the data collection process has finished. The *signature model extractor* has taken less than 15 minutes to be learned using the anomalies from two weeks.

RQ3: choice of centrality index for LOUD fault localization

I evaluated the impact of the centrality index by measuring the effectiveness of *LOUD* in localizing faults, when executing *LOUD* with the different indices for various types of injected faults. Figure 8.5 shows how the effectiveness of the localization changes over time for the five centrality indices that we identified in Chapter 6, eigenvector centrality, non-backtracking centrality, HITS algorithm for hubs and authorities, and PageRank, and for the three classes of faults that I considered, packet loss, memory leak, and CPU hog.

The diagrams reported in this and all the figures of the thesis show the F-measure computed for experiments starting from a fault injected at time 0 and lasting 120 minutes. The null value

Table 8.2. Effectiveness of the LogicModel tree (LMT) failure prediction algorithm for fault type and location

Fault type (Location)	Precision	Recall	F-Measure	Accuracy	FPR
CPU hog (Bono)	100%	93.529%	96.657%	99.8%	0%
CPU hog (Sprout)	100%	97.059%	98.507%	99.9%	0%
CPU hog (Homestead)	100%	97.041%	98.498%	99.9%	0%
CPU hog (Compute #5)	93.820%	98.817%	96.254%	99.7%	0.236%
CPU hog (Compute #7)	96.875%	91.716%	94.225%	99.6%	0.107%
Excessive workload	100%	100%	100%	100%	0%
Memory leak (Bono)	100%	98.810%	99.401%	100%	0%
Memory leak (Sprout)	100%	95.833%	97.872%	99.9%	0%
Memory leak (Homestead)	100%	96.429%	98.182%	99.9%	0%
Memory leak (Compute #5)	76.119%	91.071%	82.927%	98.7%	1.031%
Memory leak (Compute #7)	93.333%	75.000%	83.168%	98.9%	0.193%
Packet corruption (Bono)	85.973%	99.476%	92.233%	99.3%	0.669%
Packet corruption (Sprout)	87.558%	99.476%	93.137%	99.4%	0.583%
Packet corruption (Homestead)	99.429%	91.579%	95.342%	99.6%	0.022%
Packet corruption (Compute #5)	100%	100%	100%	100%	0%
Packet corruption (Compute #7)	100%	100%	100%	100%	0%
Packet latency (Bono)	96.000%	100%	97.959%	99.8%	0.173%
Packet latency (Sprout)	76.777%	84.375%	80.397%	98.4%	1.058%
Packet latency (Homestead)	72.028%	53.646%	61.493%	97.3%	0.864%
Packet latency (Compute #5)	82.857%	75.521%	79.019%	98.4%	0.648%
Packet latency (Compute #7)	62.069%	75.000%	67.925%	97.2%	1.900%
Packet loss (Bono)	100%	73.837%	84.950%	99.1%	0%
Packet loss (Sprout)	99.429%	100%	99.713%	100%	0.022%
Packet loss (Homestead)	94.152%	95.833%	94.985%	99.6%	0.215%
Packet loss (Compute #5)	100%	100%	100%	100%	0%
Packet loss (Compute #7)	82.266%	99.405%	90.027%	99.2%	0.773%
Correct execution	100%	100%	100%	100%	0%

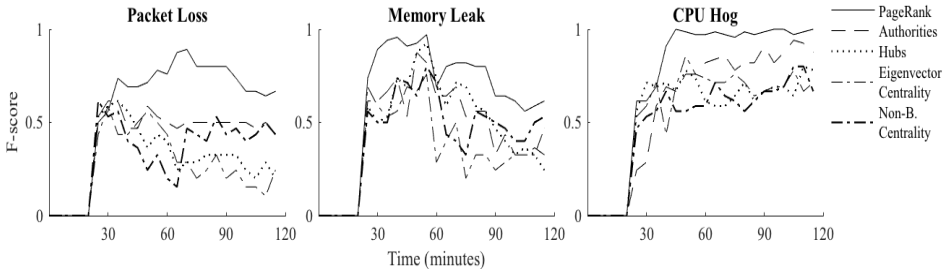

Figure 8.5. F-measure (F1-score) of each technique per fault type

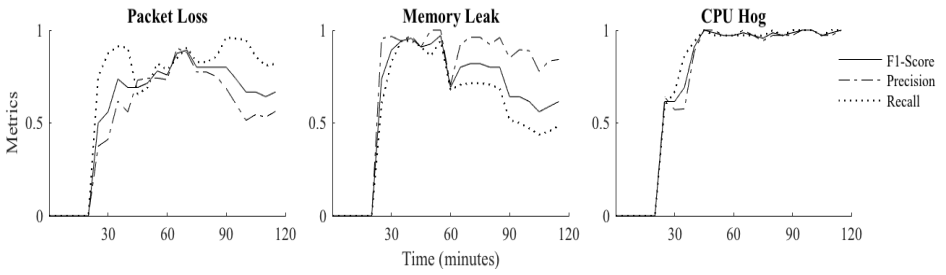

Figure 8.6. F-measure (F1-score), precision and recall of PageRank per fault type

of F-measure for the first 20 minutes of the experiments is due to *IBM ITOA-PI* that does not compute anomalies for the first 20 minutes of execution. In all the experiments with fault injection, the cloud system failed later than 120 minutes, the last time interval shown in the plots.

Figure 8.5 allows us to draw some considerations:

LOUD *performs better with PageRank than with any other considered index: LOUD* with PageRank dominates every other algorithm for all fault types through most of the observed time interval. This indicates that PageRank, and the concept of teleportation that encodes the probability of anomalies to spread non-uniformly across the graph, as it may happen in real scenarios, well captures how anomalies can spread in cloud systems.

The effectiveness of LOUD *localization depends on the fault type:* The effectiveness of fault localization strongly depends on the kind of injected fault. Indeed, anomalies spread according to significantly different patterns for different fault types. Although the centrality indices present different effectiveness for different fault types, the relative difficulty to localize a fault does not depend on the specific centrality index. Packet loss faults are the hardest faults to localize for all the indices, likely because the effect of network problems easily and quickly spreads through the system resources, in a way that is difficult to trace back to the source node. Memory leaks are also challenging but definitely easier to localize than packet loss faults. CPU hogs spread with patterns that are the easiest to trace back to the root cause among the considered faults for all algorithms.

The effectiveness of the LOUD *localization does not always improve with time:* The results do not confirm the intuition that the quality of the localization improves over time, due to the increased evidence of both the failures and their causes over time. CPU hogs are the only

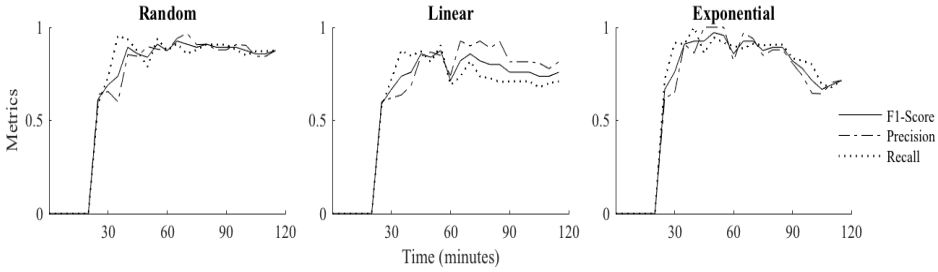

Figure 8.7. F-measure (F1-score), precision and recall of PageRank per activation pattern

type of faults with an increasing quality of the localization over time. This is likely due to the increasingly stronger impact of the CPU utilization on the affected resource compared to the other resources of the system. In the presence of both packet loss and memory leaks, the quality of the localization first increases and then decreases, with some perturbation in the intermediate phase. This suggests that in an initial phase the anomalies incrementally produced by a fault accumulate in a way that facilitates the localization task, but at some point the spreading is so extensive that it gets confused within the noise of the system, that is, with the anomalies that are regularly produced by the system despite the presence of faults.

Having identified PageRank as the index that leads to the best effectiveness compared to the other centrality indices, I conducted all the other experiments with PageRank.

RQ4: faulty resource, fault type and activation pattern for LOUD localization

I investigate the effectiveness of the *LOUD* localization with respect to three key dimensions: fault types, fault activation patterns, and faulty resources.

Figure 8.6 shows F-measure precision, and recall of *LOUD* with packet loss, memory leak and CPU hog injected faults. The precision and recall follow some interesting and complementary trends for the different classes of faults.

The localization of packet loss faults reaches a high F-measure only after a long time slack (about 65 minutes after the fault injection, and 40 minutes after the first *IBM ITOA-PI* anomaly), and does not stabilize, thus the localization often fails in identifying the faulty resource, as witnessed by a recall higher than precision. The localization of packet loss faults is precise only in a fairly short time interval (about 10 minutes) in terms of F-measure.

LOUD can reach high precision and recall memory leak faults despite they are unstable, as witnessed by an always high precision with a drop of the recall when far from the fault activation. Thus, *LOUD* may fail in identifying the fault location, but it identifies it precisely when it succeeds.

LOUD addresses well CPU hogs both in terms of precision and recall.

Figure 8.7 shows F-measure, precision and recall for the different activation patterns averaged over fault types. *LOUD* shows a similar trend for all the activation patterns, thus suggesting a reasonable independence of the effectiveness of the technique with respect to the growth rate.

The effectiveness of *LOUD* remains stable over time for both the linear and random activation patterns, and slightly decreases in the long term for the exponential pattern. This is likely due to the rapid increase and spread of anomalies that impact on the effectiveness of the localization algorithm.

Figure 8.8 shows the F-measure, precision and recall per resource, and indicates that *LOUD* can reach high precision for all the resources, slightly less stable for Homestead. A careful inspection of the results indicates that the localization phase sometime erroneously identifies Sprout as the faulty resource instead of Homestead. The architecture of the testbed discussed in Section 7.1 shows that Homestead and Sprout are highly interacting. Since *LOUD* locates well faults for other highly interacting resources, for instance Bono and Sprout, the relatively lower effectiveness for Homestead may depend on the specific characteristics of the interaction. In our prototype setting, the KPIs are unevenly distributed among Sprout and Homestead: 92 KPIs for Sprout, 71 KPIs for Homestead. Since a fault in Homestead is likely to impact on both resources, the smaller number of KPIs for Homestead than Sprout may bias the fault localization in favor of Sprout. The results might be likely improved by either collecting a set of evenly distributed KPIs or introducing a normalization strategy that takes into account the number of KPIs extracted from each resource.

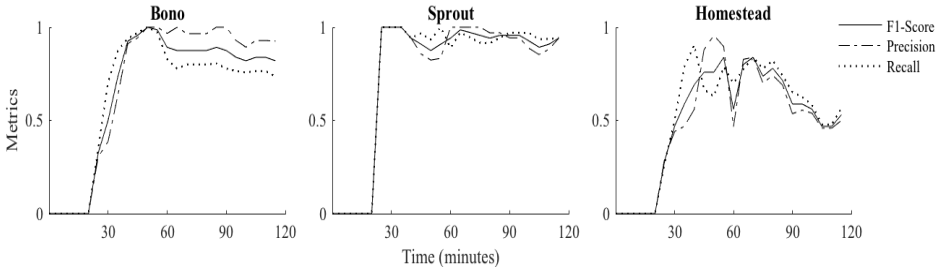

Figure 8.8. F-measure (F1-score), precision and recall of PageRank per resource

Overall, *LOUD* demonstrates to be effective in localizing faults in the cloud, with an effectiveness that depends on the kind of fault to be localized, but is largely independent from the fault activation pattern. In particular, *LOUD* is very effective with CPU hogs (the *LOUD* localization is precise, quick and constantly available after the activation of the fault), memory leaks (the *LOUD* localization is always precise, although sometime hindered by the noise in the anomalies), and useful with packet loss (the *LOUD* localization is precise, at least in some time interval after the fault activation). These results are extremely good, since the effectiveness of *LOUD* is in line with the effectiveness of competing approaches, but *LOUD* relies solely on training with normal executions, while competing approaches require long training sessions with injected faults [110]. Sessions with injected faults strongly limit the applicability of the approaches to large and evolving cloud systems that badly tolerate if at all artificially faulty sessions, and that cannot be replicated in the laboratory.

RQ5: Prediction Timeliness

I evaluated the timeliness of the prediction as the Time-To-General-Prediction (*TTGP*), the Time-To-Failure-Specific-Prediction (*TTFSP*), the Time-To-Failure for General Prediction (*TTF(GP)*) and the Time-To-Failure for Failure-Specific Prediction (*TTF(FSP)* illustrated in Figure 7.5.

In the experiments, failures correspond to either system crashes or drops of the successful SIP call rate below 60% for 5 consecutive minutes. The strong drop of call rate is mainly due to the service unavailability affected by the injected faults. Table 8.3 and Table 8.4 report the

results of the experiment. The columns *from fault occurrence to failure prediction* show the time that *PreMiSE* needed to predict a general (TTGP) and specific (TTFSP) failure, and the time that *LOUD* needed to predict a general failure, respectively. All injected faults eventually lead to a failure. Failures may occur long time after the first activation of the fault, depending on the type of injected fault and injection patterns. When failures occur outside the monitoring window (12 hours), we approximate the failure occurrence time as > 12 hours.

PreMiSE has been able to produce a general failure prediction in some minutes: 5 minutes in the best case, less than 31 minutes for most of the faults, and 65 minutes in the worst case. As having limited training inputs comparing to *PreMiSE*, *LOUD* still generates an acceptable average general prediction time of about 76 minutes. Moreover, *PreMiSE* has generated the failure specific prediction few minutes after the general prediction, with a worst case of 35 minutes from the general to the specific prediction. The readers should notice that I measure the time to prediction starting with the first activation of the seeded fault, which may not immediately lead to faulty symptoms.

The columns *From failure prediction to failure* indicate that the failures are predicted well in advance, leaving time for a manual resolution of the problem. *DyFAULT* has detected both the general and failure specific predictions at least 48 minutes before the failure for *PreMiSE*, and 28 minutes before general failure for *LOUD*. These time gaps are usually sufficient for a manual intervention. These results are also valuable for the deployment of self-healing routines, which might be activated well in advance with respect to failures.

DyFAULT predicts failure based on the analysis of IBM ITOA-PI , which works with sampling intervals of 5 minutes. *DyFAULT* could predict failures in a shorter time than 5 minutes with an anomaly detector that requires smaller sampling intervals.

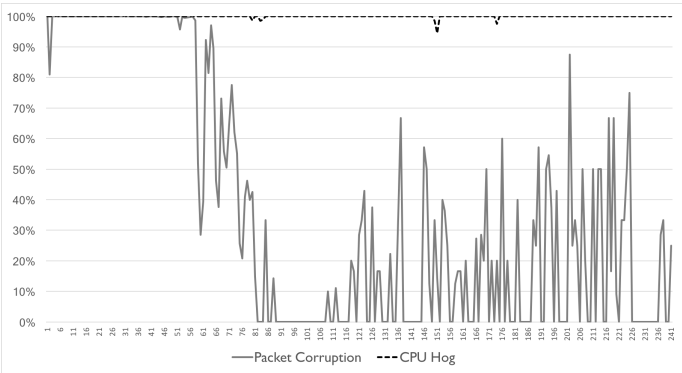

Figure 8.9. Call success rate over time

Faults of different type have very different impacts on the system, and can thus result in largely different patterns. Figure 8.9 exemplifies the different impact of faults of various types by plotting the percentage of successful calls in the experiments characterized by the longest and shortest Time-to-Failure, which correspond to CPU hog and packet corruption faults, respectively. Packet corruption faults have a gradual impact on the system, while the CPU hog faults have never caused failures in the first twelve hours of execution for the reported experiments.

Overall, *DyFAULT* demonstrated to be able to effectively predict failures, including their type well in advance to the failure time for the four classes of problems that have been investigated.

Table 8.3. PreMiSE prediction earliness for fault type and pattern

Fault Type (Pattern)	From fault occurrence to failure prediction		From failure prediction to failure	
	TTGP	*TTFSP*	*TTF (GP)*	*TTF (FSP)*
CPU hog (Random)	65 mins	80 mins	>12 hours	>12 hours
CPU hog (Constant)	45 mins	60 mins	>12 hours	>12 hours
CPU hog (Exponential)	5 mins	30 mins	>12 hours	>12 hours
Excessive workload (Random)	35 mins	50 mins	192 mins	177 mins
Excessive workload (Constant)	40 mins	55 mins	110 mins	95 mins
Excessive workload (Exponential)	30 mins	45 mins	80 mins	65 mins
Memory leak (Random)	5 mins	20 mins	55 mins	40 mins
Memory leak (Constant)	5 mins	20 mins	56 mins	41 mins
Memory leak (Exponential)	5 mins	20 mins	56 mins	41 mins
Packet corruption (Random)	30 mins	60 mins	121 mins	91 mins
Packet corruption (Constant)	30 mins	60 mins	172 mins	148 mins
Packet corruption (Exponential)	30 mins	55 mins	48 mins	23 mins
Packet latency (Random)	45 mins	70 mins	132 mins	107 mins
Packet latency (Constant)	30 mins	60 mins	132 mins	102 mins
Packet latency (Exponential)	45 mins	60 mins	59 mins	44 mins
Packet loss (Random)	50 mins	65 mins	142 mins	127 mins
Packet loss (Constant)	30 mins	65 mins	85 mins	50 mins
Packet loss (Exponential)	50 mins	65 mins	52 mins	37 mins

>12 hours indicates the cases of failures that have not been observed within 12 hours, although in the presence of active faults that would eventually lead to a system failure

Table 8.4. LOUD prediction earliness for fault type and pattern

Fault Type (Pattern)	From fault occurrence to failure prediction *TTGP*	From failure prediction to failure *TTF(GP)*
CPU hog (Random)	250 mins	>12 hours
CPU hog (Constant)	190 mins	>12 hours
CPU hog (Exponential)	145 mins	>12 hours
Excessive workload (Random)	50 mins	177 mins
Excessive workload (Constant)	50 mins	100 mins
Excessive workload (Exponential)	50 mins	60 mins
Memory leak (Random)	70 mins	80 mins
Memory leak (Constant)	50 mins	101 mins
Memory leak (Exponential)	65 mins	86 mins
Packet corruption (Random)	50 mins	101 mins
Packet corruption (Constant)	50 mins	158 mins
Packet corruption (Exponential)	50 mins	28 mins
Packet latency (Random)	50 mins	127 mins
Packet latency (Constant)	50 mins	112 mins
Packet latency (Exponential)	50 mins	54 mins
Packet loss (Random)	55 mins	137 mins
Packet loss (Constant)	50 mins	65 mins
Packet loss (Exponential)	50 mins	52 mins

>12 hours indicates the cases of failures that have not been observed within 12 hours, although in the presence of active faults that would eventually lead to a system failure

RQ6: Comparative Evaluation

I compare *DyFAULT* to *IBM ITOA-PI* to estimate the improvement of *DyFAULT* with respect to state-of-the-art data analytics approaches. Table 8.5 reports the precision, recall, F-Measure, accuracy and false positive rate of *PreMiSE* , *LOUD* and *IBM ITOA-PI* .

IBM ITOA-PI infers the threshold of normal performance for KPI values, and raises alarms only for persistent anomalies, that is, if the probability that a KPI is anomalous for 3 of the last 6 intervals is above a certain threshold value [62]. Columns IBM ITOA-PI *(alarms)* and IBM ITOA-PI *(anomalies)* of Table 8.5 report all and the persistent anomalies that *IBM ITOA-PI* detects and signal, respectively. In both cases *IBM ITOA-PI* is less effective than *DyFAULT*: *IBM ITOA-PI* does not raise any alarm, thus failing to predict the failure (recall = 0%, precision and F-measure not computable), and records far too many anomalies, thus signalling all potential failures (recall of 100%) diluted in myriads false alarms (false positive rate = 100%). In a nutshell, *IBM ITOA-PI* reports every legal executions as a possible failure, and relies on system administrators to react to the alarms. *PreMiSE* is effective: The high values of the five measures indicate that *PreMiSE* and *LOUD* predicts most failures with a negligible amount of false positives. In summary, the combination of anomaly detection and signature-based analysis proposed in *DyFAULT* largely outperforms *IBM ITOA-PI* .

Table 8.5. Comparative evaluation of DyFAULT and IBM ITOA-PI

Measures	***PreMiSE***	*LOUD*	***IBM ITOA-PI* (alarms)**	***IBM ITOA-PI* (anomalies)**
Precision	100%	86.9565%	–	94.118%
Recall	98.751%	100%	0%	100%
F-Measure	99.372%	93.023%	–	96.970%
Accuracy	98.797%	92.5%	5.882%	94.118%
FPR	0%	15%	0%	100%

RQ7: Overhead

DyFAULT interacts directly with the system only with the *KPI Monitor*, which in our prototype implementation collects the KPIs by means of the SNMPv2c monitoring service for Clearwater [22], the Ceilometer telemetry service for OpenStack [123] and a Linux OS agent that I implemented for Ubuntu. All other computation is performed on a dedicated node, and does not impact on the overall performance of the target system. Thus, the *DyFAULT* overhead on the running system is limited to the overhead of the monitoring services that I expect to be very low.

Figure 8.10 reports cpu, memory, disk and network consumption when the system is executed with and without the monitoring infrastructure. The experimental results shown in the figure confirm a limited impact of the overhead on the overall system, with a an absolute increment of relative increment of 2.63% of CPU usage (a relative increment of 18%) and 1.91% of memory usage (a relative increment of .04%). They also indicate a negligible impact of the monitoring infrastructure on disk (measured as read and written bytes) and network (measured as number of sent and received packets) usage, accounting for few hundreds bytes over tens of thousands and few packets over thousands, respectively. These results are perfectly compatible with systems with strong performance requirements, such as telecommunication infrastructures.

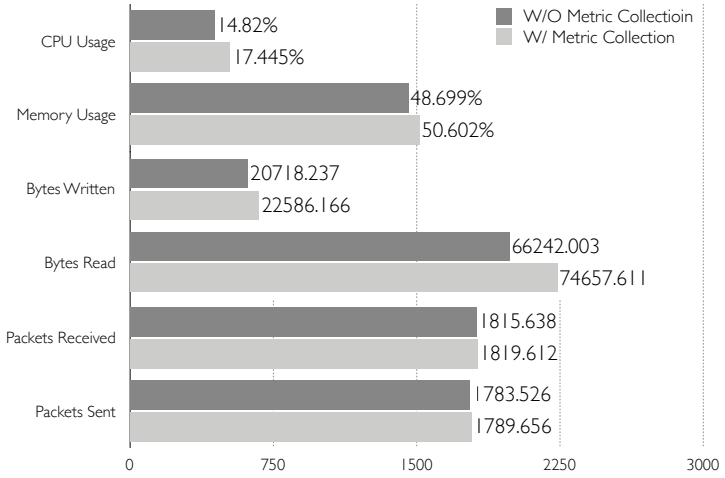

Figure 8.10. PreMiSE overhead

Threats to Validity

I briefly discuss the main threats to the validity of the result reported in this thesis.

Threats to the external validity The main threat to the external validity of the results derive from the limited scope of the experiments. In this chapter, I report the results of a through set experiments on date gathered from a IP Multimedia Subsystem (IMS) service running on a physically distributed multi-tier cloud system, with a configuration commonly used in industrial settings, a traffic profile that marches a typical commercial profile and a set of seeded faults that correspond to the most relevant faults for the considered type of system. The good results that I obtained by experimenting with a system may not immediately generalize to systems of different type, different traffic profiles and different types of faults. We mitigated this threat to validity by choosing a widely used system executed on a common cloud infrastructure, with faults of types commonly present in such systems, and with diverse fault patterns. We also mitigate this threat by providing the experimental results as a publicly available replication package.

Threats to the internal validity The main threat to the internal validity of the results derive from the implementation of the experimental prototype. We mitigated this threat by carefully designing and testing both the experimental system and the *DyFAULT* prototype, by repeating the experiments several time in different conditions to assure the coherence of the results, and by comparing the results with independent sources, when available.

Chapter 9

Conclusions

Cloud computing reduces costs by improving resource utilization efficiency, with a considerable amount of complexity and dynamics that challenge the reliability of the system. Pre-deployment testing and analysis of cloud applications cannot prevent runtime failures that can heavily impact on the quality and costs of cloud services. Self-healing approaches tackle runtime failures, by early identifying failing context and taking actions to avoid failures, or at least mitigate their impact. Self-healing approaches rely on mechanisms to early predict failures and localize faults. The new challenges of cloud systems motivate a new holistic self-healing approach, which must be accurate, lightweight and proactive, to ensure reliable cloud applications. Self-healing techniques work at runtime, thus they offer automatic and flexible ways to increase reliability by detecting errors, diagnosing errors, and either fixing the errors or mitigating their effects. Self-Healing Systems leverage the time between the activation of a fault and the failure by taking actions to avoid failures. Self-Healing systems shall predict failures, localize the faults and fix or mask them before the failure occurrence.

In this thesis, I present *DyFAULT*, an approach that focus on failure prediction and fault localization. *DyFAULT* predicts failures by detecting anomalous systems states early enough to diagnose the causing errors and fix them before the failure occurrence, and localizes faults by leveraging the collected data to pinpoint the location of error and possibly the type of the fault. The main contribution of this thesis are *PreMiSE*, an approach to accurately predict failures and localize faults that requires training with fault seeding, *LOUD*, an approach to predict failures and localize faults that requires training with data from normal execution only, a prototype implementation of the two approaches, a set of experimental results that show advantages and limitation of the proposed approaches.

The work presented in this dissertation opens few interesting research directions towards optimizing the amount of KPIs that shall be monitored for accurate predictions and localizations, and substantially reducing training time to address dynamic cloud system that support auto-scaling and live migration.

Bibliography

[1] Abreu, R., Zoeteweij, P. and v. Gemund, A. J. C. [2009]. Spectrum-based multiple fault localization, *2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE'09, pp. 88–99.

[2] Arcuri, A. and Yao, X. [2008]. A novel co-evolutionary approach to automatic software bug fixing, *Proceedings of IEEE Congress on Evolutionary Computation*, CEC '08, IEEE Computer Society, pp. 162–168.

[3] Arnold, A., Liu, Y. and Abe, N. [2007]. Temporal causal modeling with graphical granger methods, *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '07, ACM, pp. 66–75.

[4] Avizienis, A., Laprie, J.-C., Randell, B. and Landwehr, C. [2004]. Basic concepts and taxonomy of dependable and secure computing, *IEEE Transactions on Dependable Secure Computing* **1**(1): 11–33.

[5] Baresi, L. and Guinea, S. [2007]. Dynamo and self-healing bpel compositions, *Companion to the Proceedings of the 29th International Conference on Software Engineering*, ICSE COMPANION '07, pp. 69–70.
URL: *http://dx.doi.org/10.1109/ICSECOMPANION.2007.31*

[6] Baresi, L., Guinea, S. and Pasquale, L. [2007]. Self-healing BPEL processes with dynamo and the jboss rule engine, *Proceedings of the 2007 International Workshop on Engineering of Software Services for Pervasive Environments*, ESSPE '07, pp. 11–20.

[7] Bauer, E. and Adams, R. [2012]. *Reliability and Availability of Cloud Computing*, Wiley.

[8] Bennett, C. [2015]. Chaos Monkey, `https://github.com/Netflix/SimianArmy/wiki/Chaos-Monkey`.

[9] Blohowiak, A., Basiri, A., Hochstein, L. and Rosenthal, C. [2016]. A platform for automating chaos experiments, *Proceedings of the International Symposium on Software Reliability Engineering Workshops*, ISSREW '16, IEEE Computer Society, pp. 5–8.

[10] Box, G. E. P., Jenkins, G. and Gregory, R. [2008]. *Time Series Analysis: Forecasting and Control*, Wiley.

[11] Buyya, R., Calheiros, R. N. and Li, X. [2012]. Autonomic cloud computing: Open challenges and architectural elements, *Proceedings of the International Conference on Emerging Applications of Information Technology*, EAIC '12, IEEE Computer Society, pp. 3–10.

[12] Cabral, B. and Marques, P. [2011]. A transactional model for automatic exception handling, *Computer Languages, Systems and Structures* **37**(1): 43–61.

[13] Candea, G., Kawamoto, S., Fujiki, Y., Friedman, G. and Fox, A. [2004]. Microreboot — a technique for cheap recovery, *Proceedings of the Symposium on Operating Systems Design and Implementation*, OSDI'04, USENIX Association, pp. 31–44.

[14] Carzaniga, A., Gorla, A., Mattavelli, A., Pezzè, M. and Perino, N. [2013]. Automatic recovery from runtime failures, *Proceedings of the International Conference on Software Engineering*, ICSE '13, IEEE Computer Society, pp. 782–791.

[15] Carzaniga, A., Gorla, A., Perino, N. and Pezzè, M. [2010]. Automatic workarounds for web applications, *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, ACM, pp. 237–246.

[16] Carzaniga, A., Gorla, A., Perino, N. and Pezzè, M. [2015]. Automatic workarounds: Exploiting the intrinsic redundancy of web applications, *ACM Transactions on Software Engineering and Methodologies* **24**(3): 16.

[17] Carzaniga, A., Gorla, A. and Pezzè, M. [2008a]. Healing web applications through automatic workarounds, *International Journal on Software Tools for Technology Transfer (STTT)* **10**(6): 493–502.

[18] Carzaniga, A., Gorla, A. and Pezzè, M. [2008b]. Self-healing by means of automatic workarounds, *SEAMS '08: Proceedings of the 5th International Workshop on Software Engineering for Adaptive and Self-Managing Systems*, ACM, pp. 17–24.

[19] Casanova, P., Garlan, D., Schmerl, B. and Abreu, R. [2013]. Diagnosing architectural run-time failures, *2013 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pp. 103–112.

[20] Casanova, P., Garlan, D., Schmerl, B. and Abreu, R. [2014]. Diagnosing unobserved components in self-adaptive systems, *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS 2014, Association for Computing Machinery, New York, NY, USA, pp. 75–84.

[21] Casanova, P., Schmerl, B., Garlan, D. and Abreu, R. [2011]. Architecture-based run-time fault diagnosis, *Proceedings of the 5th European Conference on Software Architecture*, ECSA'11, Springer-Verlag, Berlin, Heidelberg, pp. 261–277.

[22] Case, J. D., McCloghrie, K. and Rose, M.and Waldbusser, S. [1996]. Introduction to community-based snmpv2, IETF RFC 1901.

[23] Castro, M., Rodrigues, R. and Liskov, B. [2003]. Base: Using abstraction to improve fault tolerance, *ACM Trans. Comput. Syst.* **21**(3): 236–269.
URL: *http://doi.acm.org/10.1145/859716.859718*

[24] Chandola, V., Banerjee, A. and Kumar, V. [2009]. Anomaly detection: A survey, *ACM Computing Surveys* **41**(3): 15.

[25] Chen, G., Jin, H., Zou, D., Zhou, B. B., Qiang, W. and Hu, G. [2010]. Shelp: Automatic self-healing for multiple application instances in a virtual machine environment, *Proceedings of the International Conference on Cluster Computing*, CLUSTER '10, IEEE Computer Society, pp. 97–106.

[26] Chen, J. and Shang, W. [2017]. An exploratory study of performance regression introducing code changes, *Proceedings of the IEEE International Conference on Software Maintenance and Evolution*, ICSME '17, IEEE Computer Society, pp. 341–352.

[27] Chen, M. Y., Kiciman, E., Fratkin, E., Fox, A. and Brewer, E. [2002]. Pinpoint: Problem determination in large, dynamic internet services, *Proceedings of the International Conference on Dependable Systems and Networks*, DSN '02, IEEE Computer Society, pp. 595–604.

[28] Cheng, B. H., de Lemos, R., Giese, H., Inverardi, P., Magee, J., Andersson, J., Becker, B., Bencomo, N., Brun, Y., Cukic, B. et al. [2009]. Software engineering for self-adaptive systems: A research roadmap, *Software engineering for self-adaptive systems*, Springer, pp. 1–26.

[29] Cheng, S.-W., Garlan, D., Schmerl, B., Steenkiste, P. and Hu, N. [2002]. Software architecture-based adaptation for grid computing, *HPDC' 02: Proceedings of 11th IEEE International Symposium on High Performance Distributed Computing*, pp. 389–398.

[30] Cherkasova, L., Ozonat, K., Mi, N., Symons, J. and Smirni, E. [2008]. Anomaly? application change? or workload change? towards automated detection of application performance anomaly and change, *Proceedings of the International Conference on Dependable Systems and Networks*, DSN '08, IEEE Computer Society, pp. 452–461.

[31] Cid-Fuentes, J. A., Szabo, C. and Falkner, K. [2018]. Adaptive performance anomaly detection in distributed systems using online svms, *IEEE Transactions on Dependable Secure Computing* .

[32] Dallmeier, V., Zeller, A. and Meyer, B. [2009]. Generating fixes from object behavior anomalies, *Proceedings of the International Conference on Automated Software Engineering*, ASE '09, IEEE Computer Society.

[33] Danne, C., Duck, V., Kloepper, B. and Tichy, M. [2007]. Considering runtime restrictions in self-healing distributed systems, *21st International Conference on Advanced Information Networking and Applications (AINA '07)*, pp. 228–235.

[34] Dashofy, E. M., Van der Hoek, A. and Taylor, R. N. [2002]. Towards architecture-based self-healing systems, *Proceedings of the Workshop on Self-healing Systems*, WOSS '02, ACM, pp. 21–26.

[35] Dean, D. J., Nguyen, H. and Gu, X. [2012]. Ubl: Unsupervised behavior learning for predicting performance anomalies in virtualized cloud systems, *Proceedings of the International Conference on Autonomic Computing*, ICAC '12, ACM, pp. 191–200.

[36] Dean, D. J., Nguyen, H., Gu, X., Zhang, H., Rhee, J., Arora, N. and Jiang, G. [2014]. Perfscope: Practical online server performance bug inference in production cloud computing infrastructures, *Proceedings of the Annual Symposium on Cloud Computing*, SoCC '14, ACM, pp. 8:1–8:13.

[37] Dean, D. J., Nguyen, H., Wang, P., Gu, X., Sailer, A. and Kochut, A. [2015]. Perfcompass: Online performance anomaly fault localization and inference in infrastructure- as-a-service clouds, *IEEE Transactions on Parallel and Distributed Systems* **PP**(99): 1–1.

[38] Demsky, B., Zhou, J. and Montaz, W. [2010]. Recovery tasks: an automated approach to failure recovery, *RV '10: Proceedings of the First International Conference on Runtime Verification*, pp. 229–244.
URL: *http://portal.acm.org/citation.cfm?id=1939399.1939420*

[39] Dobson, G. [2006]. Using WS-BPEL to implement software fault tolerance for Web services, *Proceedings of the EUROMICRO Conference on Software Engineering and Advanced Applications*, SEAA '06, IEEE Computer Society, pp. 126–133.

[40] Duan, S. and Babu, S. [2008]. Guided problem diagnosis through active learning, *Proceedings of the International Conference on Autonomic Computing* pp. 45–54.

[41] Elnozahy, M., Alvisi, L., Wang, Y.-m. and Johnson, D. B. [2002]. A survey of rollback-recovery protocols in message-passing systems, *ACM Computing Surveys* **34**(3): 375–408.

[42] Engel, M. and Freisleben, B. [2005]. Supporting autonomic computing functionality via dynamic operating system kernel aspects, *AOSD '05: Proceedings of the 4th International Conference on Aspect-Oriented Software Development*, pp. 51–62.

[43] Foo, K. C., Jiang, Z. M., Adams, B., Hassan, A. E., Zou, Y. and Flora, P. [2015]. An industrial case study on the automated detection of performance regressions in heterogeneous environments, *Proceedings of the International Conference on Software Engineering*, ICSE '15, IEEE Computer Society.

[44] Friedman, J., Hastie, T., Tibshirani, R. et al. [2000]. Additive logistic regression: a statistical view of boosting, *Annals of Statistics* **95**(2): 337–407.

[45] Fu, S. and Xu, C.-Z. [2007]. Exploring event correlation for failure prediction in coalitions of clusters, *Proceedings of the Annual International Conference on Supercomputing*, SC '07, IEEE Computer Society, pp. 1–12.

[46] Fulp, E. W., Fink, G. A. and Haack, J. N. [2008]. Predicting computer system failures using support vector machines., *Proceedings of the USENIX conference on Analysis of system logs*, WASL'08, USENIX Association, pp. 5–5.

[47] Garg, S., Huang, Y., Kintala, C. and Trivedi, K. S. [1996]. Minimizing completion time of a program by checkpointing and rejuvenation, *SIGMETRICS Performance Evaluation Review* **24**(1): 252–261.

[48] Gashi, I., Popov, P., Stankovic, V. and Strigini, L. [2004]. On designing dependable services with diverse off-the-shelf SQL servers, *Architecting Dependable Systems II*, Vol. 3069 of *Lecture Notes in Computer Science*, Springer, pp. 191–214.

[49] Gaudin, B., Vassev, E. I., Nixon, P. and Hinchey, M. [2011]. A control theory based approach for self-healing of un-handled runtime exceptions, *Proceedings of the 8th ACM international conference on Autonomic computing*, ICAC '11, ACM, New York, NY, USA, pp. 217–220.
URL: *http://doi.acm.org/10.1145/1998582.1998633*

[50] Ghaith, S., Wang, M., Perry, P. and Murphy, J. [2013]. Profile-based, load-independent anomaly detection and analysis in performance regression testing of software systems, *Proceedings of the European Conference on Software Maintenance and Reengineering*, CSMR '13.

[51] Gorla, A., Mariani, L., Pastore, F., Pezzè, M. and Wuttke, J. [2010]. Achieving cost-effective software reliability through self-healing, *Computing and Informatics* **29**(1): 93–115.

[52] Gregory F., C. and Edward, H. [1992]. A bayesian method for the induction of probabilistic networks from data, *Machine Learning* **9**(4): 309–347.

[53] Grottke, M., Li, L., Vaidyanathan, K. and Trivedi, K. [2006]. Analysis of software aging in a web server, *IEEE Transactions on Reliability* **55**(3): 411–420.

[54] Grottke, M. and Trivedi, K. S. [2007]. Fighting bugs: Remove, retry, replicate, and rejuvenate, *IEEE Computer* **40**(2): 107–109.

[55] Guo, C., Yuan, L., Xiang, D., Dang, Y., Huang, R., Maltz, D. A. D., Liu, Z., Wang, V., Pang, B., Chen, H., Lin, Z.-W. and Kurien, V. [2015]. Pingmesh: A large-scale system for data center network latency measurement and analysis, *Proceedings of the ACM SIGCOMM Conference*, SIGCOMM '15, ACM, pp. 139–152.

[56] Guo, P. J. [2011]. Sloppy python: Using dynamic analysis to automatically add error tolerance to ad-hoc data processing scripts, *WODA '11: Proceedings of the 2011 International Workshop on Dynamic Analysis*.

[57] Herodotou, H., Ding, B., Balakrishnan, S., Outhred, G. and Fitter, P. [2014]. Scalable near real-time failure localization of data center networks, *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, SIGKDD '14, ACM, pp. 1689–1698.

[58] Hosek, P. and Cadar, C. [2013]. Safe software updates via multi-version execution, *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, IEEE Press, Piscataway, NJ, USA, pp. 612–621.
URL: *http://dl.acm.org/citation.cfm?id=2486788.2486869*

[59] Huang, Y., Kintala, C., Kolettis, N. and Fulton, N. D. [1995]. Software rejuvenation: Analysis, module and applications, *Proceedings of the 25th International Symposium on Fault-Tolerant Computing*, FTCS '95, IEEE Computer Society, pp. 381–390.

[60] Huebscher, M. C. and McCann, J. A. [2008]. A survey of autonomic computing – degrees, models, and applications, *ACM Computing Surveys* **40**(3): 7:1–7:28.

[61] Ibidunmoye, O., Hernández-Rodriguez, F. and Elmroth, E. [2015]. Performance anomaly detection and bottleneck identification, *ACM Computing Surveys* **48**(1): 4:1–4:35.

[62] IBM [2014]. *SmartCloud Analytics - Predictive Insights 1.3 (Tutorial)*. Document Revision R2E2.

[63] IBM Corporation [n.d.]. IBM SmartCloud Analytics - Predictive Insights, `https://developer.ibm.com/itoa/docs/ibm-operations-analytics/`. Last access: oct 2017.

[64] Jadeja, Y. and Modi, K. [2012]. Cloud computing-concepts, architecture and challenges, *Proceedings of the International Conference on Computing, Electronics and Electrical Technologies*, ICCEET '12, IEEE Computer Society, pp. 877–880.

[65] Jayathilaka, H., Krintz, C. and Wolski, R. M. [2018]. Detecting performance anomalies in cloud platform applications, *IEEE Transactions on Cloud Computing* .

[66] John, G. H. and Langley, P. [1995]. Estimating continuous distributions in bayesian classifiers, *Proceedings of the Conference on Uncertainty in Artificial Intelligence*, UAI'95, Morgan Kaufmann Publishers, pp. 338–345.

[67] Johnsson, A., Meirosu, C. and Flinta, C. [2014]. Online network performance degradation localization using probabilistic inference and change detection, *Proceedings of the Conference on Network Operations and Management Symposium*, NOMS '14, IEEE Computer Society, pp. 1–8.

[68] Jones, J. A. and Harrold, M. J. [2005]. Empirical evaluation of the tarantula automatic fault-localization technique, *ASE '05: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*.

[69] Kandula, S., Mahajan, R., Verkaik, P., Agarwal, S., Padhye, J. and Bahl, P. [2009]. Detailed diagnosis in enterprise networks, *Proceedings of the ACM SIGCOMM Computer Communication Review* **39**(4): 243–254.

[70] Kang, H., Chen, H. and Jiang, G. [2010]. Peerwatch: A fault detection and diagnosis tool for virtualized consolidation systems, *Proceedings of the International Conference on Autonomic Computing*, ICAC '10, ACM, pp. 119–128.

[71] Kavulya, S., Daniels, S., Joshi, K., Hiltunen, M., Gandhi, R. and Narasimhan, P. [2012]. Draco: Statistical diagnosis of chronic problems in large distributed systems, *Proceedings of the International Conference on Dependable Systems and Networks*, DSN '12, IEEE Computer Society, pp. 1–12.

[72] Khanna, R., Wallace, E., Brar, J., Marr, M., McKelvie, S., DeSantis, P., Nowland, I., Klein, M., Mason, J. and Gabrielson, J. [2014]. Monitoring and detecting causes of failures of network paths. US Patent 8,661,295.
URL: *https://www.google.com/patents/US8661295*

[73] Kleinberg, J. M. [1999]. Authoritative sources in a hyperlinked environment, *J. ACM* **46**(5): 604–632.

[74] Kohavi, R. [1995]. The power of decision tables, *Proceedings of the European Conference on Machine Learning*, ECML'95, Springer, pp. 174–189.

[75] Kumar, K. P. and Naik, N. S. [2014]. Self-healing model for software application, *International Conference on Recent Advances and Innovations in Engineering (ICRAIE-2014)*, pp. 1–6.

[76] Landwehr, N., Hall, M. and Frank, E. [2005]. Logistic model trees, *Machine Learning* **59**(1-2): 161–205.

[77] Langville, A. N. and Meyer, C. D. [2005]. A survey of eigenvector methods for web information retrieval, *SIAM Review* **47**(1): 135–161.

[78] Le Goues, C., Dewey-Vogt, M., Forrest, S. and Weimer, W. [2012]. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each, *Proceedings of the International Conference on Software Engineering*, ICSE '12, IEEE Computer Society, pp. 3–13.

[79] Leite, A. F., Alves, V., Rodrigues, G. N., Tadonki, C., Eisenbeis, C. and Melo, A. C. M. A. D. [2016]. Autonomic provisioning, configuration, and management of inter-cloud environments based on a software product line engineering method, *2016 International Conference on Cloud and Autonomic Computing (ICCAC)*, pp. 72–83.

[80] Looker, N., Munro, M. and Xu, J. [2005]. Increasing Web service dependability through consensus voting, *Proceedings of the International Computer Software and Applications Conference*, COMPSAC '05, IEEE Computer Society, pp. 66–69.

[81] Malik, H., Hemmati, H. and Hassan, A. E. [2013]. Automatic detection of performance deviations in the load testing of Large Scale Systems, *Proceedings of the International Conference on Software Engineering*, ICSE '13, IEEE Computer Society, pp. 1012–1021.

[82] Mariani, L., Pastore, F. and Pezzè, M. [2011]. Dynamic analysis for diagnosing integration faults, *IEEE Transactions on Software Engineering* .

[83] Martin, T., Zhang, X. and Newman, M. E. J. [2014]. Localization and centrality in networks, *Phys. Rev. E* **90**: 052808.

[84] Mell, P. and Grance, T. [2009]. The nist definition of cloud computing, *National Institute of Standards and Technology* **53**(6): 50.

[85] Menasce, D., Gomaa, H., s. Malek and Sousa, J. [2011]. Sassy: A framework for self-architecting service-oriented systems, *IEEE Software* **28**(6): 78–85.

[86] Mi, H., Wang, H., Zhou, Y., Lyu, M. and Cai, H. [2013]. Toward fine-grained, unsupervised, scalable performance diagnosis for production cloud computing systems, *IEEE Transactions on Parallel and Distributed Systems* **24**(6): 1245–1255.

[87] Modafferi, S., Mussi, E. and Pernici, B. [2006]. SH-BPEL: a self-healing plug-in for WS-BPEL engines, *Proceedings of the 1st Workshop on Middleware for Service Oriented Computing*, MW4SOC '06, ACM, pp. 48–53.

[88] Mosincat, A. and Binder, W. [2008]. Transparent runtime adaptability for BPEL processes., *Proceedings of the International Conference on Service Oriented Computing*, ICSOC '08, Springer, pp. 241–255.

[89] Moura Silva, L., Alonso, J., Silva, P., Torres, J. and Andrzejak, A. [2007]. Using virtualization to improve software rejuvenation, *Proceedings of the IEEE International Symposium on Network Computing and Applications*, NCA '07, IEEE Computer Society, pp. 33–44.

[90] Mysore, R. N., Mahajan, R., Vahdat, A. and Varghese, G. [2014]. Gestalt: Fast, unified fault localization for networked systems, *Proceedings of the USENIX Conference on Annual Technical Conference*, ATC '14, USENIX Association, pp. 255–267.

[91] Nistor, A. and Ravindranath, L. [2014]. Suncat: Helping developers understand and predict performance problems in smartphone applications, *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA '14, ACM, pp. 282–292.

[92] Novaković, D., Vasić, N., Novaković, S., Kostić, D. and Bianchini, R. [2013]. Deepdive: Transparently identifying and managing performance interference in virtualized environments, *Proceedings of the USENIX Conference on Annual Technical Conference*, ATC '13, USENIX Association, pp. 219–230.

[93] Omer Angel, J. F. and Hoory, S. [2014]. The non-backtracking spectrum of the universal cover of a graph, *Transactions of the American Mathematical Society* **367**(6): 4287–4318.

[94] Ozcelik, B. and Yilmaz, C. [2016]. Seer: A Lightweight Online Failure Prediction Approach, *IEEE Transactions on Software Engineering* **42**(1): 26–46.

[95] Pagano, D., Juan, M. A., Bagnato, A., Roehm, T., Bruegge, B. and Maalej, W. [2012]. Fastfix: Monitoring control for remote software maintenance, *ICSE*, pp. 1437–1438.

[96] Pecchia, A., Cotroneo, D., Kalbarczyk, Z. and Iyer, R. K. [2011]. Improving log-based field failure data analysis of multi-node computing systems, *Proceedings of the International Conference on Dependable Systems and Networks*, DSN '11, IEEE Computer Society, pp. 97–108.

[97] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V. et al. [2011]. Scikit-learn: Machine learning in python, *Journal of machine learning research* **12**(Oct): 2825–2830.

[98] Perkins, J. H., Kim, S., Larsen, S., Amarasinghe, S., Bachrach, J., Carbin, M., Pacheco, C., Sherwood, F., Sidiroglou, S., Sullivan, G., Wong, W.-F., Zibin, Y., Ernst, M. D. and Rinard, M. [2009]. Automatically patching errors in deployed software, *Proceedings of the Symposium on Operating Systems Principles*, SOSP '09, pp. 87–102.

[99] Platt, J. et al. [1998]. Fast training of support vector machines using sequential minimal optimization, *Advances in Kernel Methods: Support Vector Learning*, MIT Press, pp. 185–208.

[100] Project Clearwater [n.d.]. IMS in the cloud, `http://www.projectclearwater.org`. Last access: may 2015.

[101] Qiang, G., Ziming, Z. and Song, F. [2010]. Ensemble of bayesian predictors for autonomic failure management in cloud computing, *Proceedings of the International Conference on Computer Communications and Networks*, ICCCN '10, IEEE Computer Society, pp. 1–6.

[102] Qin, F., Tucek, J., Sundaresan, J. and Zhou, Y. [2005]. Rx: Treating bugs as allergies—a safe method to survive software failures, *Proceedings of the 20th ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '05, pp. 235–248.

[103] Richard, G. and Olivier, J. [n.d.]. SIPp, `http://sipp.sourceforge.net`. Last access: may 2015.

[104] Riungu, L. M., Taipale, O. and Smolander, K. [2010]. Research issues for software testing in the cloud, *Proceedings of the International Conference on Cloud Computing Technology and Science*, IEEE Computer Society, pp. 557–564.

[105] Roy, S. and Feamster, N. [2013]. Characterizing correlated latency anomalies in broadband access networks, *Proceedings of the ACM SIGCOMM Computer Communication Review*, CCR '13, IEEE Computer Society, pp. 525–526.

[106] Salfner, F., Lenk, M. and Malek, M. [2010]. A survey of online failure prediction methods, *ACM Computing Surveys* **42**(3): 1–42.

[107] Salfner, F., Schieschke, M. and Malek, M. [2006]. Predicting failures of computer systems: A case study for a telecommunication system, *Proceedings of the International Parallel and Distributed Processing Symposium*, IPDPS '06, IEEE Computer Society, pp. 8–pp.

[108] Samimi, H., Aung, E. D. and Millstein, T. [2010]. Falling back on executable specifications, *ECOOP '10: Proceedings of the 24th European Conference on Object-Oriented Programming*, pp. 552–576.
URL: *http://portal.acm.org/citation.cfm?id=1883978.1884015*

[109] Sauvanaud, C., Lazri, K., Kaaniche, M. and Kanoun, K. [2016]. Anomaly detection and root cause localization in virtual network functions, *Proceedings of the International Symposium on Software Reliability Engineering*, ISSRE '16, IEEE Computer Society.

[110] Sauvanaud, C., Lazri, K., Kaâniche, M. and Kanoun, K. [2016]. Anomaly detection and root cause localization in virtual network functions, *Proceedings of the International Symposium on Software Reliability Engineering*, ISSRE '16, IEEE Computer Society, pp. 196–206.

[111] Scott, J. P. and Carrington, P. J. [2011]. *The SAGE Handbook of Social Network Analysis*, Sage Publications Ltd.

[112] Sharma, A. B., Chen, H., Ding, M., Yoshihira, K. and Jiang, G. [2013]. Fault detection and localization in distributed systems using invariant relationships, *Proceedings of the International Conference on Dependable Systems and Networks*, DSN '13, IEEE Computer Society, pp. 1–8.

[113] Sharma, B., Jayachandran, P., Verma, A. and Das, C. R. [2013]. Cloudpd: Problem determination and diagnosis in shared dynamic clouds, *Proceedings of the International Conference on Dependable Systems and Networks*, DSN '13, IEEE Computer Society, pp. 1–12.

[114] Shen, K., Stewart, C., Li, C. and Li, X. [2009]. Reference-driven performance anomaly identification, *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '09, ACM, pp. 85–96.

[115] Stack, P., Xiong, H., Mersel, D., Makhloufi, M., Terpend, G. and Dong, D. [2017]. Self-healing in a decentralised cloud management system, *Proceedings of the International Workshop on Next Generation of Cloud Architectures*, CloudNG '17, ACM, pp. 3:1–3:6.

[116] Subramanian, S., Thiran, P., Narendra, N. C., Mostefaoui, G. K. and Maamar, Z. [2008]. On the enhancement of BPEL engines for self-healing composite web services, *Proceedings of the International Symposium on Applications and the Internet*, SAINT '08, IEEE Computer Society, pp. 33–39.

[117] Taher, Y., Benslimane, D., Fauvet, M.-C. and Maamar, Z. [2006]. Towards an approach for Web services substitution, *Proceedings of the International Database Engineering and Applications Symposium*, IDEAS '06, IEEE Computer Society, pp. 166–173.

[118] Tan, J., Pan, X., Marinelli, E., Kavulya, S., Gandhi, R. and Narasimhan, P. [2010]. Kahuna: Problem diagnosis for mapreduce-based cloud computing environments, *Proceedings of the Conference on Network Operations and Management Symposium*, NOMS '10, IEEE Computer Society, pp. 112–119.

[119] Tan, Y., Gu, X. and Wang, H. [2010]. Adaptive system anomaly prediction for large-scale hosting infrastructures, *Proceedings of the Symposium on Principles of Distributed Computing*, PODC '12, ACM, pp. 173–182.

[120] Tati, S., Ko, B. J., Cao, G., Swami, A. and La Porta, T. F. [2015]. Adaptive algorithms for diagnosing large-scale failures in computer networks, *IEEE Transactions on Parallel and Distributed Systems* **26**(3): 646–656.

[121] Tetali, S. D. [2015]. *Program Analyses for Cloud Computations*, PhD thesis, University of California, Los Angeles.

[122] The KVM Project [n.d.]. Kernel based virtual machine, `http://www.linux-kvm.org`. Last access: may 2015.

[123] The OpenStack Project [n.d.a]. Ceilometer, `https://wiki.openstack.org/wiki/Ceilometer`. Last access: may 2015.

[124] The OpenStack Project [n.d.b]. Open source software for creating private and public clouds, `https://www.openstack.org`. Last access: may 2015.

[125] Thorat, P., Raza, S. M., Nguyen, D. T., Im, G., Choo, H. and Kim, D. S. [2015]. Optimized self-healing framework for software defined networks, *Proceedings of the 9th International Conference on Ubiquitous Information Management and Communication*, IMCOM '15, ACM, New York, NY, USA, pp. 7:1–7:6.

[126] Traeger, A., Deras, I. and Zadok, E. [2008]. Darc: Dynamic analysis of root causes of latency distributions, *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '08, ACM, pp. 277–288.

[127] University of Waikato [n.d.]. Weka 3: Data mining software in java, `http://www.cs.waikato.ac.nz/ml/weka/`. Last access: may 2015.

[128] Vilalta, R. and Ma, S. [2002]. Predicting rare events in temporal domains, *Proceedings of the International Conference on Data Mining*, ICDM '02, IEEE Computer Society, pp. 474–481.

[129] Wang, T., Zhang, W., Ye, C., Wei, J., Zhong, H. and Huang, T. [2016]. Fd4c: Automatic fault diagnosis framework for web applications in cloud computing, *IEEE Transactions on Systems, Man, and Cybernetics: Systems* **46**(1): 61–75.

[130] Wang, Y.-M., Huang, Y., Vo, K.-P., Chung, P.-Y. and Kintala, C. [1995]. Checkpointing and its applications, *Proceedings of the 25th International Symposium on Fault-Tolerant Computing*, FTCS '95, pp. 22–31.

[131] Wei, Y., Pei, Y., Furia, C. A., Silva, L. S., Buchholz, S., Meyer, B. and Zeller, A. [2010]. Automated fixing of programs with contracts, *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA '10, ACM, pp. 61–72.

[132] Weimer, W., Nguyen, T., Goues, C. L. and Forrest, S. [2009]. Automatically finding patches using genetic programming, *Proceedings of the International Conference on Software Engineering*, ICSE '09, IEEE Computer Society, pp. 364–374.

[133] Williams, A. W., Pertet, S. M. and Narasimhan, P. [2007]. Tiresias: Black-box failure prediction in distributed systems, *Proceedings of the International Parallel and Distributed Processing Symposium*, IPDPS '07, IEEE Computer Society, pp. 1–8.

[134] Witten, I. H., Frank, E. and Hall, M. A. [2011]. *Data Mining: Practical Machine Learning Tools and Techniques*, Morgan Kaufmann Publishers.

[135] Wong, W. E., Gao, R., Li, Y., Abreu, R. and Wotawa, F. [2016]. A survey on software fault localization, *IEEE Transactions on Software Engineering* **42**(8): 707–740.

[136] Wu, W., Wang, G., Akella, A. and Shaikh, A. [2013]. Virtual network diagnosis as a service, *Proceedings of the Annual Symposium on Cloud Computing*, SoCC '13, ACM, pp. 9:1—-9:15.

[137] Xu, X., Zhu, L., Weber, I., Bass, L. and Sun, D. [2014]. Pod-diagnosis: Error diagnosis of sporadic operations on cloud applications, *Proceedings of the International Conference on Dependable Systems and Networks*, DSN '14, IEEE Computer Society, pp. 252–263.

[138] Yongmin, T., Hiep, N., Zhiming, S., Xiaohui, G., Chitra, V. and Deepak, R. [2012]. Prepare: Predictive performance anomaly prevention for virtualized cloud systems, *Proceedings of the International Conference on Distributed Computing Systems*, ICDCS '12, IEEE Computer Society, pp. 285–294.

[139] Yuan, C., Lao, N., Wen, J.-R., Li, J., Zhang, Z., Wang, Y.-M. and Ma, W.-Y. [2006]. Automated known problem diagnosis with event traces, *Proceedings of the ACM SIGOPS EuroSys European Conference on Computer Systems*, EUROSYS '06, ACM, pp. 375–388.

[140] Zavou, A., Portokalidis, G. and Keromytis, A. D. [2012]. Self-healing multitier architectures using cascading rescue points, *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, ACM, New York, NY, USA, pp. 379–388.

[141] Zhang, H., Jiang, L. and Su, J. [2005]. Hidden naive bayes, *Proceedings of the National Conference on Artificial Intelligence*, AAAI'05, AAAI Press, pp. 919–924.

[142] Zhang, S., Cohen, I., Goldszmidt, M., Symons, J. and Fox, A. [2005]. Ensembles of models for automated diagnosis of system performance problems, *Proceedings of the International Conference on Dependable Systems and Networks*, DSN '05, IEEE Computer Society, pp. 644–653.

[143] Zhang, X., Meng, F., Chen, P. and Xu, J. [2016]. Taskinsight: A fine-grained performance anomaly detection and problem locating system, *Proceedings of the International Conference on Cloud Computing*, CLOUD '16, USENIX Association, pp. 917–920.