A Survey of AIOps Methods for Failure Management

PAOLO NOTARO, Chair of Computer Architecture and Parallel Systems, Technical University of Munich, Germany and Huawei Munich Research Center, Germany
JORGE CARDOSO, Department of Informatics Engineering/CISUC, University of Coimbra, Portugal and Huawei Munich Research Center, Germany
MICHAEL GERNDT, Chair of Computer Architecture and Parallel Systems, Technical University of Munich, Germany

Modern society is increasingly moving toward complex and distributed computing systems. The increase in scale and complexity of these systems challenges O&M teams that perform daily monitoring and repair operations, in contrast with the increasing demand for reliability and scalability of modern applications. For this reason, the study of automated and intelligent monitoring systems has recently sparked much interest across applied IT industry and academia. Artificial Intelligence for IT Operations (AIOps) has been proposed to tackle modern IT administration challenges thanks to Machine Learning, AI, and Big Data. However, AIOps as a research topic is still largely unstructured and unexplored, due to missing conventions in categorizing contributions for their data requirements, target goals, and components. In this work, we focus on AIOps for Failure Management (FM), characterizing and describing 5 different categories and 14 subcategories of contributions, based on their time intervention window and the target problem being solved. We review 100 FM solutions, focusing on applicability requirements and the quantitative results achieved, to facilitate an effective application of AIOps solutions. Finally, we discuss current development problems in the areas covered by AIOps and delineate possible future trends for AI-based failure management.

CCS Concepts: • **Computing methodologies** → *Artificial intelligence*; • **Applied computing** → *Service-oriented architectures*; **Data centers**; • **Computer systems organization** → *Dependable and fault-tolerant systems and networks;*

Additional Key Words and Phrases: AIOps, IT operations and maintenance, failure management, artificial intelligence

ACM Reference format:
Paolo Notaro, Jorge Cardoso, and Michael Gerndt. 2021. A Survey of AIOps Methods for Failure Management. *ACM Trans. Intell. Syst. Technol.* 12, 6, Article 81 (November 2021), 45 pages.
https://doi.org/10.1145/3483424

Authors' addresses: P. Notaro, Chair of Computer Architecture and Parallel Systems, Technical University of Munich, Boltzmannstr. 3, Garching b. München, Bavaria, Germany, 85748 and Huawei Munich Research Center, Riessstr. 25, Munich, Bavaria, Germany, 80992; email: paolo.notaro@tum.de; J. Cardoso, Department of Informatics Engineering/CISUC, University of Coimbra, Portugal and Huawei Munich Research Center, Riessstr. 25, Munich, Bavaria, Germany, 80992; email: jorge.cardoso@huawei.com; M. Gerndt, Chair of Computer Architecture and Parallel Systems, Technical University of Munich, Boltzmannstr. 3, Garching b. München, Bavaria, Germany, 85748; email: gerndt@in.tum.de.

2157-6904/2021/11-ART81 $15.00
https://doi.org/10.1145/3483424

1 INTRODUCTION

Society is nowadays highly reliant on critical **Information Technology (IT)** infrastructures. To satisfy the technological needs required by organizations to conduct their business, IT systems have grown larger and more complex. Aims for higher scalability and efficient resource utilization have also moved attention toward decentralized systems, introducing additional layers of abstraction and complexity. Moreover, due to their criticality in day-to-day operations, IT infrastructures of today are required to have higher standards of availability and reliability to confront modern challenges such as the Internet of Things, 5G, Autonomous Driving, and Smart Cities. System failures can cause a significant amount of disruption during the normal operation of an IT service, which can rapidly turn into customer dissatisfaction. Because of the very nature of these new distributed systems, their administration has become more difficult and prone to the appearance of failures and performance issues. **Operations and Maintenance (O&M)** teams are particularly strained by the scale and complexity of modern systems, where the large quantity and multimodality of monitoring data to oversee in real-time can easily be overwhelming even to specialized IT operators. Unexpected faults also cause IT operators to stop their monitoring operations and devote their time to solve the encountered issues. At the same time, the new dimension of modern systems raises the question of how to allocate and efficiently distribute all kinds of resources (computational, physical, and energetic) in large-scale shared computing environments.

Current O&M solutions have strong limitations in providing reliable and scalable IT services. They heavily rely on the manual intervention of human operators, who can no longer cope with appearing issues rapidly and efficiently due to the large scale of modern systems. Typical human intervention windows span several hours, compared to the minute-scale intervention of automated systems. Current automated approaches, however, often exhibit other limitations, especially in terms of applicability (as they cannot adapt to new problems and situations) and scalability (as they cannot handle large data volumes in real-time). For this reason, in recent years much research interest has been directed toward developing intelligent software systems to tackle O&M problems effectively, grouped under the term **Artificial Intelligence for IT Operations (AIOps)**. AIOps investigates the use of **Artificial Intelligence (AI)** for the automated management of IT services [17, 36, 74, 75, 79, 96, 107, 116, 129, 133]. AIOps helps SRE, DevOps, and O&M teams enhance the quality and reliability of IT service offerings, by using intelligent algorithms and the large quantity of data available thanks to monitoring infrastructures. AIOps relies on data-driven technologies (Machine Learning, Big Data, Data Mining, Analytics, and Visualization) to observe the operational status of the infrastructure, minimize the impact of failures during day-to-day operations, and manage the allocation of computer resources proactively. Compared to traditional approaches, AIOps is:

- *fast*, because it reacts independently and automatically to real-time problems, without requiring long manual debugging and analysis sessions;
- *efficient*, because it takes advantage of the monitoring infrastructure holistically, removing data silos and improving issue visibility. By forecasting workload requirements and modeling request patterns, AIOps improves resource utilization, identifies performance bottlenecks, and reduces wastages. By relieving IT operators from investigation and repair burdens, AIOps enables in-house personnel to concentrate more effort on other tasks;
- *effective*, because it allocates computer resources proactively and can offer a large set of actionable insights for root-cause diagnosis, failure prevention, fault localization, recovery, and other O&M activities.

Several companies have started to deliver AIOps tools as products over the past few years [17, 96, 116, 129, 133], while several tech leaders have adopted AIOps algorithms to maintain

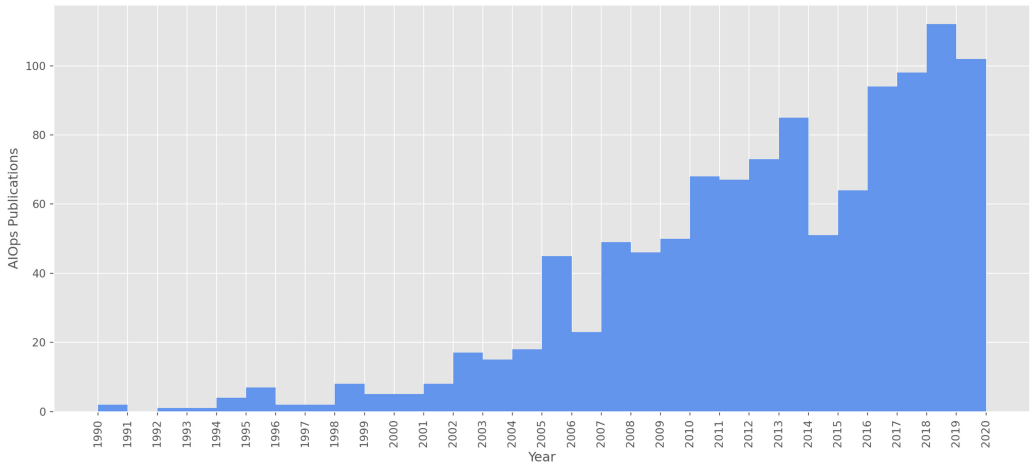

Fig. 1. Total number of publications related to AIOps analyzed in this survey by year of publication.

their on-premise computing facilities efficiently [36, 75, 79, 82]. In the academic context, several research groups are taking advantage of the recent advances in **Machine Learning (ML)** and AI to explore open problems related to AIOps, such as online failure prediction [27, 79, 148], or anomaly detection [22, 107, 150]. Despite these recent advances, the concept of using AI for improving IT offerings is not new. Starting in the mid-1970s, different computer scientists have proposed to locate software faults in source code using statistical models and code complexity metrics [16, 19, 28, 53, 66, 90]. Since the early 1990s, several online software [34, 56, 65, 134] and hardware [58, 100, 101] failure prediction models have been proposed. Other failure prevention methods also date back to the same period [5, 48, 49, 55, 57, 135, 136]. Other areas of AIOps, like anomaly detection, event correlation, or problem identification, have attested contributions for at least 20 years [2, 15, 20, 24, 40, 68, 144]. Therefore, the reliability and efficiency of computer systems have always been a research focus since the birth of Internet-based services. Rather than a steady flow of research contributions, what we observe is an increasing trend reaching a highpoint in recent years [113] (see Figure 1), motivated also by the rise in interest toward AI and Big Data technologies during the last decade.

Because AIOps is a modern concept bridging research and industry, there is no widely adopted definition yet [36]. The term AIOps was originally coined by Gartner in 2017 [74], for which "AIOps platforms utilize big data, modern Machine Learning and other advanced analytics technologies to directly and indirectly enhance IT operations." Some sources have adopted this definition [75, 129, 133], while others have complemented it or have proposed a different vision on the topic [17, 36, 79, 96, 107, 116]. Table 1 provides an overview of the available sources defining AIOps. All these definitions share two common objectives: (1) enhance IT services for customers and (2) provide full visibility and understanding into the operational state of IT systems. Some definitions also highlight the importance of data collection [74, 75], real-time automated operation [36, 96], and scalability [17, 116].

The most cited AIOps problems are anomaly detection [36, 75, 107, 129], root-cause analysis [17, 75, 96, 116], resource provisioning [36, 107, 116, 129], failure remediation [17, 96, 116], and failure prediction and prevention [17, 75, 79]. We can observe how a significant fraction of these classes treats the occurrence of *failures*. We can then divide AIOps into two macro-areas: failure management and resource provisioning. Failure management deals with the appearance of failures,

Table 1. Available AIOps Definitions with Corresponding Mentions of Related Technologies and Focuses

Source(s)	Definition	Technologies	Focus
Gartner [74] Levin et al. [75] BMC [129] Resolve [133]	"collect logs, traces and telemetry data, and analyze the collected data to [...] directly and indirectly enhance IT operations [...] identify patterns in monitoring, capacity, service desk, and automation data [...] the application of machine learning (ML) and data science to IT operations problems"	Big Data, ML, Data Analytics	Data collection, Real-time understanding
Broadcom [17]	"establish proactive, automated remediation capabilities that help IT teams deliver superior customer experiences"	ML, Data Science	Issue visibility, Automation, CI/CD
Dang et al. [36] OpsRamp [116] Li et al. [79]	"efficiently and effectively build and operate services that are easy to support and maintain [...] solving everyday IT operational problems at scale [...] including intelligent alerting, alert correlation, alert escalation, auto-remediation, root-cause analysis and capacity optimization [...] assist DevOps engineers to improve the quality of computing platforms in a cost-effective manner"	AI, ML, Network Science, Combinatorial Optimization	Real-time understanding, CI/CD, Scale and efficiency
Moogsoft [96] Nedelkoski et al. [107]	"provide full visibility into the state and performance of the IT systems [...] give IT operations teams a real-time understanding of any issues affecting the availability or performance [...] replace a broad range of IT Operations tasks including availability, performance, and monitoring of service [...] enable detection of faults and issues of services"	AI, ML, Big Data	Real-time understanding, Issue visibility

in all available means and possible windows of interventions, both in dynamic aspects (such as the runtime behavior of the system or the network responsiveness) and static aspects (like the source code or the datacenter configuration). Rather than a homogeneous and well-defined research area, it is a wide and heterogeneous space of contributions coming from different specialized areas. Resource provisioning optimizes the allocation of the diverse set of resources necessary to provide IT services, such as power, computation time, network bandwidth and virtual memory.

Being a recent and cross-disciplinary field, AIOps is still a largely unstructured area of study. The existing contributions are scattered across different conferences, apply different terms for the same concepts, or use the same terms for different concepts. Moreover, the high number of application areas renders the search and collection of relevant papers difficult. The additional scarcity of previous comparison studies and surveys motivates the need for a comprehensive study, able to collect, categorize and summarize past contributions, to facilitate the comparison and sharing of all available approaches for each O&M task.

In our previous work [113], we conducted a systematic mapping study to analyze and categorize the whole spectrum of past AIOps contributions, to delineate a taxonomy of topics and open problems, and to draw quantitative results about publications by topic and time. In this article, we concentrate on one of the two identified macro-areas, i.e., failure management, to analyze and organize contributions according to target problems and applicability. This enables our readers to:

- gather a comprehensive overview of failure management in AIOps, by the definition of a taxonomy of approaches and the description of the current landscape of problems;
- obtain a reference index for solutions of problems against requirements. Requirements expressed in the form of available data sources and target high-level components (source code, software, hardware, network, and datacenter) can be mapped to a list of available solutions thanks to the indexing made available in our analysis;
- understand the current state of failure management, which problems have been long investigated and which require new research. To this end, we also present an outline of possible future directions for the fields of AIOps and failure management.

The remainder of this article is organized as follows. Section 2 describes previous related review studies in AIOps and failure management. Section 3 presents the methodology of this work, summarizing the planning choices of systematic mapping study used to identify the papers later presented, as well as the terminology and metric conventions. Section 4 delineates the structure of failure management in AIOps and presents a selection of papers divided in thematic sections, describing their line of work, contribution, approach, input source, target component, and application field. Section 5 summarizes the results and outcomes drawn from our discussion.

2 RELATED WORK

As the AIOps areas mentioned above are large both in number and range of applications, it is reasonable to expect numerous works focusing on filtering and categorizing best approaches and practices. Several other surveys and mapping studies have been in fact conducted in the areas covered by AIOps [22, 24, 39, 46, 47, 60, 63, 70, 71, 91, 99, 106, 109, 111, 112, 122, 125, 130, 139, 146, 152]. However, no previous work has provided an updated, comprehensive review of AIOps approaches for failure management.

Table 2 summarizes the most relevant survey and systematic review contributions regarding IT Operations and Artificial Intelligence, organized by main topic and other focuses. Most works treat single tasks [22, 24, 39, 60, 63, 91, 106, 111, 112, 122, 139, 152] or general goals [46, 47, 125, 130, 146] inside AIOps, which are specific to particular intervention methods.

A second category of works [70, 71, 99, 109] treat failure management integrally, but some of the works inside this group are outdated and do not reflect the current progress of the field, while the most recent works do not focus on AI-based approaches or do not offer a comprehensive list of contributions. The closest match to our analysis is represented by the work of Mukwevho and Celik [99], who present a survey on fault management in cloud systems. Differently from them, we do not focus on any particular computing system, and we focus on the manifestation of faults (i.e., errors and failures) rather than root causes (see Section 3.3 for terminology). We also choose to integrate AI and Machine learning approaches in the conventional scheme of failure management approaches, rather than treating them in a separate category. All these considerations, including the observations about the missing structure and terminology conventions presented in Section 1, motivate the need for an in-depth study in this area, like the one here presented.

3 METHODOLOGY

3.1 Systematic Mapping Study

A **systematic mapping study (SMS)** was conducted to obtain relevant and representative literature in the field of AIOps. Different from a **systematic literature review (SLR)**, the ultimate goal of a systematic mapping study is to provide an overview of a specific research area, to obtain a set of related papers, and to delineate trends present inside such area [67]. Relevant papers are collected via well-defined search and selection criteria, while research trends are identified using categorization schemes covering different aspects, e.g., such main topic, origin, or type of contribution. We choose this instrument because we are interested in gathering contributions for the survey and obtaining insights into the field, such as the distribution of works in different AIOps subareas and the temporal evolution of the interest toward specific topics. An in-depth discussion of the mapping study methodology, the categorization scheme, and the selection strategy of contribution is available in a work separately published [113], accessible online.[1]

[1] https://arxiv.org/abs/2012.09108.

Table 2. Related AI Surveys and Systematic Mapping Studies (SMS) Conducted in Areas Covered by this Work

Ref.	Year	Type	Main Topic	Focuses
IT Operations				
[71]	2007	Survey	IT Operations	AI, Operational Research
[70]	2011	Survey	IT Operations	AI, Operational Research
[109]	2013	SMS	Failure Management	Temporal & Geographical Trends, Service Level, Others
[99]	2018	Survey	Failure Management	Cloud Computing
Failure Prevention				
[112]	2011	Survey	Failure Prevention	Combinatorial Testing
[39]	2015	Survey	Failure Prevention/Detection	Software
[106]	2016	Survey	Fault Injection	Software
[91]	2017	Survey	Software Defect Prediction	Machine Learning, PROMISE dataset
Failure Prediction				
[125]	2007	Survey	Failure Prediction	AI, Integrated Systems
[152]	2007	Survey	Failure Prediction	Clusters
[122]	2010	Survey	Failure Prediction	Online Methods
[63]	2019	Survey	Failure Prediction	High Performance Computing
Failure Detection				
[60]	2015	Survey	Anomaly Detection	Bottleneck Identification
[24]	2009	Survey	Anomaly Detection	-
[22]	2019	Survey	Anomaly Detection	Deep Learning
[139]	2013	Survey	Anomaly Detection	Network
[111]	2008	Survey	Internet Traffic Classification	Machine Learning, IP Networks
Root-cause Analysis (RCA)				
[130]	2017	Survey	Root-cause Analysis	-
[146]	2016	Survey	Fault Localization	Software
[46]	2015	Survey	Fault Diagnosis	Model- and signal-based approaches in Industrial Systems
[47]	2015	Survey	Fault Diagnosis	Knowledge-based and hybrid approaches in Industrial Systems

Thanks to our mapping study, we collected 1,086 AIOps contributions and inferred an AIOps taxonomy based on thematic areas and commonly treated tasks (see Figure 2). Our taxonomy groups contributions in the two main macro-areas: failure management and resource provisioning. Each macro-area divides contributions into different categories, based on end-goals and target problems. We also classify the relevant papers according to the following categorization aspects: target components, input data sources, AI methods (see Tables 4 and 8).

For the failure management macro-area, the focus of our survey, we also divide approach categories into proactive and reactive, based on the window of intervention (red box in Figure 2). Moreover, we introduce a second level of categorization for failure management, which divides each category into several subcategories based on the specific target problem solved. Examples of subcategories are software defect prediction for failure prevention (Section 4.1.1) and log enhancement for failure detection (Section 4.3.3). For our survey discussion, we select 100 most prominent contributions in failure management from the total result set of 1,086, covering all categories and subcategories defined for this field of study. For the exhaustive list of papers covered, divided into categories and subcategories, see Table 4 in the next section.

3.2 Evaluation Metrics

In our analysis, we provide quantitative results for the papers under investigation. This section provides an overview of the evaluation metrics employed for comparison throughout the survey discussion.

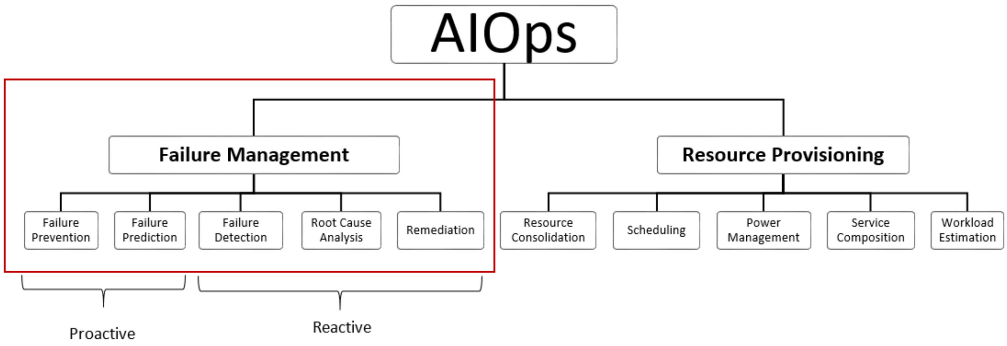

Fig. 2. Taxonomy of AIOps as observed in the identified contributions. In the red box, the focus of this survey.

Table 3. Contingency Table for Prediction Tasks

		Predicted Class	
		Positive Class	**Negative Class**
Real Class	**Positive Class**	True Positives (TP)	False Negatives (FN)
	Negative Class	False Positives (FP)	True Negatives (TN)

For scalar prediction (or regression) tasks, a widely adopted metric is the ***Mean-squared Error* (MSE)**, defined as the average squared difference between target and predicted values:

$$MSE = \frac{1}{N}\sum_{i}^{N}\left(y_i^{pred} - y_i\right)^2. \tag{1}$$

A measure adopted across all classification problems (software defect prediction, root-cause diagnosis, recovery, etc.) is *accuracy*, i.e., the ratio of classified samples assigned to the correct class. In some contexts, however, accuracy may appear as a misleading metric to evaluate the quality of prediction. This is the case, for example, for problems with a high predominance of one class, where trivial models can be constructed to reach high accuracy just exploiting data skewness. A similar consideration applies to detection problems (analyzed, e.g., in Sections 4.2 and 4.3), where the positive class, i.e., the detected failure, may appear less frequently than the negative class, even though it constitutes the most critical aspect from an evaluation point of view. In such cases, it is common to adopt more representative measures, derived from the notion of contingency table [122]:

Using this convention, *accuracy* can be written as

$$ACC = \frac{TP + TN}{TP + TN + FP + FN}. \tag{2}$$

To quantify the ability of a predictor to identify positive samples correctly, the *precision* measure is usually employed, while to measure the ability of a detector to report true positive samples, the *recall* measure (also known as *true positive rate* or *sensitivity*) is used. They are defined as follows:

$$P = \frac{TP}{TP + FP}, \qquad R = \frac{TP}{TP + FN}. \tag{3}$$

Moreover, the ***false-positive rate*** (**FPR**, also called false alarm date), which identifies the proportion of wrongly reported failures, is defined as follows:

$$FPR = \frac{FP}{FP + TN}. \tag{4}$$

Since precision and recall do not take into account the number of true negatives, some papers compare results in terms of ***true-negative rate*** (**TNR**, or specificity) and recall. The true-negative rate is defined as follows:

$$TNR = \frac{TN}{TN + FP}. \tag{5}$$

Precision and recall can often be traded off with each other by adjusting sensitivity thresholds inside algorithms, so that an increase in precision can be obtained by reducing recall and vice versa. One possibility to evaluate both measures at the same time is to use the *F1-score* (or F-score/F-measure), computed as the harmonic mean of precision and recall:

$$F_1 = 2 \cdot \frac{P \cdot R}{P + R}. \tag{6}$$

A final possibility is to use **receiver operating characteristic (*ROC*)** curves, parametric line plots which describe the variation of two metrics in relation to changes in the sensitivity threshold. Precision-recall curves are possible, but it is common to plot the recall against the false-positive rate. From this type of curves the **Area under the ROC curve (*AUCROC*)** measure can be computed. A higher AUCROC score then indicates a better classifier.

3.3 Terminology

In our discussion, we also adopt a variety of error-related terms such as fault, failure and root cause. From a terminology point of view, we adopt the convention of Salfner et al. [122] for the characterization of faulty behavior. According this convention:

- *errors* are deviations from the correct system state;
- *failures* are manifestations of undesired deviations in the delivery of a service;
- *faults* (or root causes) are the primary causes of undesired behavior (i.e., the errors).

Moreover, we often encounter different terms related to data sources that may have an ambiguous meaning depending on the context, such as logs or traces. To be consistent in our discussion, we group the observed data sources according to this convention:

- *source code* represents any unit of software source code used as input to a prediction system, independently of the form and extension (e.g., function, file, module, class, etc.);
- *testing resources* comprise tools used to perform in- and post-release software debugging, in particular unit test suites, execution profiles or run description reports;
- *system metrics* measure various numerical quantities at the hardware, OS, software and environment level, describing resource utilization and the overall process state of the system;
- ***key performance indicators*** **(KPI)** provide information about the status of services and the associated requirements that need to be met during runtime operations. They quantitatively measure the quality of served requests with parameters such as latency, uptime, failure rate, availability, and so on;
- *network traffic* is the collection of network packets exchanged over Internet by different hosts. It includes the payload and control information such as ports, addresses, protocol standards and other parameters;
- *topology* is any information describing the spatial relations inside a working system, when used as a input;

- *incident reports* are collected with the help of service desk and internal problem management systems to identify common problems and facilitate resolution. Usually, they describe the problem with text and categorical attributes and they may be associated to a resolution team or routing sequence;
- *event logs* (or simply *logs*) are collections of human-interpretable printing statements describing software events occurring in runtime operations. They are typically stored as independent files and log entries (i.e., lines) are associated to a predefined format (or log key);
- *execution (distributed) traces* are hierarchical descriptions of the modules and services invoked to satisfy a user request. They are usually annotated with the service name or category and the time duration of each module (called *span*).

4 AI APPROACHES IN FAILURE MANAGEMENT

Failure Management (FM) is the study of techniques deployed to minimize the appearance and impact of failures. In large-scale systems failures are inevitable, so adequate protection mechanisms need to be put up to minimize their and satisfy **Service-level Objectives (SLO)** [26]. Following an established convention [10, 21, 99, 119, 160], we differentiate between proactive (failure avoidance) and reactive (failure tolerance) approaches (differently from Reference [99], we do not endorse the "resilient approaches" category for AI approaches). In our hierarchical scheme, failure avoidance comprises any approach aiming to address failures in anticipation of their occurrence, either by predicting the appearance of errors based on the system state, or by taking preventive actions to minimize their future incidence. Failure avoidance is divided into failure prevention (Section 4.1) and online failure prediction (Section 4.2). Failure tolerance techniques, however, deal with errors after their appearance to assist humans and improve **mean-time-to-recovery (MTTR)**. Failure tolerance includes failure detection (Section 4.3), root-cause analysis (Section 4.4), and remediation (Section 4.5).

The five categories above mentioned divide FM based on the temporal window of intervention. In our previous work [113], we quantified the frequency of FM publications in our five categories. The FM area with the highest number of contributions is failure detection (226, 33.7%), followed by root-cause analysis (179, 26.7%) and online failure prediction (177, 26.4%). However, failure prevention and remediation are the areas with the smallest number of attested contributions (71 and 17, respectively). These trends are confirmed and even more accentuated when we look at the most recent publication period (2018 onwards): only 11 (5) publications have been found for failure prevention (remediation) in this time frame.

Within each of these five categories, we can group current contributions based on targets problems (or *tasks*) that a contribution aims to solve (e.g., failure prevention includes software defect prediction, fault injection, software rejuvenation, and checkpointing). Table 4 categorizes the FM contributions analyzed in this work by category, task and AI methods used.

While we consider our five-category scheme for FM comprehensive and stable over time (although the relative proportion of works in each category may change), we envision that landscape of tasks and target problems within each category may evolve more quickly, by expanding some of the current minority tasks and by presenting new problems and challenges in the years to come (see discussion in Section 5.1). Table 8 (at the end of the section) present the same list of works grouped by task, AI method, employed data sources, and high-level target components.

4.1 Failure Prevention

An important (and yet no so common) possibility to deal with failures proactively is failure prevention, treated in this section. Failure prevention mechanisms, tend to minimize the occurrence and impact of failures, by analyzing the configuration of the system, both in static aspects (like the

Table 4. Papers Analyzed in this Survey by Category, Task, and AI Method

Category	Task	AI Method	
Failure Prevention	Software Defect Prediction (SDP)	Linear Models: [35, 52, 102, 117, 153] Tree-based: [38, 50, 94, 98, 141] Bayesian Networks: [38, 50, 115, 141] Logistic Regression: [35, 98, 103, 141]	Naive Bayes: [38, 94, 98, 141] SVM: [42, 50] Others: [76, 102, 153]
	Fault Injection	Clustering: [105, 128] Tree-based: [105]	Linear Models: [128]
	Software Aging and Rejuvenation	Markov Models: [21, 136, 137] Linear Models: [4, 21, 49]	Tree-based: [4]
	Checkpointing	Markov Models: [62, 95, 114]	
Online Failure Prediction	Hardware Failure Prediction	Tree-based: [77, 78, 89, 104, 138, 148] Neural Networks: [37, 78, 89, 149, 165, 167] SVM: [89, 164, 167] Markov Models: [161, 164]	Naive Bayes: [54, 88] Clustering: [54, 101] Logistic Regression: [89] Others: [33, 143]
	System Failure Prediction	SVM: [44, 81, 160] Bayesian Networks: [31, 119] Markov Models: [123]	Autoregressive Models: [23, 119] Neural Networks: [61, 160] Others: [81, 160]
Failure Detection	Anomaly Detection	Clustering: [11, 45, 72, 127] Autoencoders: [7, 131, 150, 159] Markov Models: [110, 127] PCA: [72, 73, 151] Other Neural Networks: [18, 41, 45, 92, 162]	PCFG: [11, 26] FSM: [12, 45] Tree-based: [86, 151] Others: [29, 31, 86, 87, 127]
	Internet Traffic Classification	Neural Networks: [8, 142] Naive Bayes: [97]	SVM: [43]
	Log Enhancement	Tree-based: [168]	Others: [163]
Root-cause Analysis	Fault Localization	Graph Mining: [9, 30, 156] Others: [1, 72, 83, 85, 110, 121, 145]	Search: [30, 80, 132]
	Root-cause Diagnosis	Pattern Matching: [6, 127] Bayesian Networks: [31, 64]	Others: [25, 26, 64, 124, 154]
	RCA - Others	Clustering: [32, 84, 120] Logistic Regression: [14, 120]	Others: [3, 14, 32]
Remediation	Incident Triage	Markov Models: [126]	Bayesian Decision Theory: [158]
	Solution Recommendation	Text Analysis: [82, 140, 166]	Similarity-based: [140]
	Recovery	Markov Models: [124]	

source code) and dynamic aspects (e.g., the availability of computing resources in physical hosts). The common goal is to take or suggest preventive actions; however, the strategies to achieve this end goal vary extensively in targets and mode of application. Moreover, we can distinguish preventive operations in a offline setting and an online setting. For the former, we observe a large predominance of software debugging techniques, usually categorized under the name of **software defect prediction (SDP)**, determined to evaluate failure risks from source code analysis; fault injection techniques are also sporadically present, with the objective of stress-testing the system to gain additional insights and prevent future failures. In an online setting, we observed methods dealing with the problem of software aging, categorized under the name of software rejuvenation; and checkpointing techniques, to deliver efficient restart strategies in the presence of irreversible errors.

4.1.1 Software Defect Prediction. SDP determines the probability of running into a software bug (or defect) within a functional unit of code (i.e., a function, a class, a file, or a module). The fundamental assumption, which connects SDP to the occurrence of faults, is the observation that defect-prone code generates failures when executed. Therefore, estimating the remaining density of bugs in a functional unit of code allows software maintainers to prioritize their efforts, concentrating on the most vulnerable modules, files, and methods.

A traditional method to identify fault-prone software consists in constructing defect predictors from code metrics. Code metrics are handcrafted features obtained directly from source code, which potentially have the power to predict fault proneness in software. Over the last decades, different groups of code metrics have been proposed: the original module-level metrics, designed during the 1970s for procedural languages, describing graph complexity (McCabe metrics [90]) and reading complexity (Halstead metrics [53]); Chidamber and Kemerer [28] later defined a suite of metrics for object-oriented software on a class level, usually referred to as "CK metrics" (from the authors' initials); Briand et al. [16] have introduced coupling metrics as a measure of interconnection between software modules; finally, the **Lines-of-Code (LOC)** metric is widely adopted across the research community.

Nagappan et al. [102] investigate an SDP approach employing code complexity metrics. They however argue that multicollinearity (or inter-correlation) between some of these metrics render the problem more complex. They apply **Principal Component Analysis (PCA)** to obtain a smaller set of uncorrelated features. Then, they construct linear regression models to predict post-release defects in five different datasets. They also attempt to apply models learned on one project onto other projects to test cross-project applicability, obtaining mixed results, and therefore arguing for the similarity of projects as a requirement for transfer learning.

In Reference [94], Menzies et al. discuss the use of static code metrics (like McCabe and Halstead's), arguing in their favor and rejecting common claims, such as the one of being uninformative (because of cross-correlation of values within the code feature set) or suboptimal compared to other features. In particular, they argue for their practicality, popularity, and usefulness. Attention is moved from the choice of metrics to the choice of learners: by applying decision trees and Naïve Bayes models, software modules are classified as defect-prone or defect-free. The authors argue in favor of Naïve Bayes with the use of logarithmic features, which obtains the best detection performance (71% recall and 25% false alarm rate).

Elish et al. [42] use **Support Vector Machines (SVM)** for software defect prediction. Failure-proneness is predicted at the function level, from a composite set of McCabe, Halstead, and LOC features. The approach applies Gaussian kernels to enable non-linear modeling and is tested on four software projects from the NASA MDP dataset [93]. A comparison with other Machine Learning methods (logistic regression, multilayer perceptron, k-NN, k-means, **Bayesian Networks (BN)**, Naïve Bayes, Random Forest, and Decision Tree) is carried out, where the SVM approach outperforms all the models in terms of recall (≥ 0.994), at the expense of a lower precision (≥ 0.8495). SVMs exhibit accuracies ($\geq$0.8459) and F-scores ($\geq$0.916) comparable to the other methods.

Another common class of methods for SDP is constituted by BN, due to their positive qualities such as accuracy, interpretability, and modifiability. Dejaeger et al. [38] consider 15 different BN classifiers, including Naïve Bayes as a special case. The approach is validated on two datasets (the already mentioned NASA MDP [93] and the Eclipse project repository [157]) and it is compared to other Machine Learning algorithms using AUC and H-measure. The results show how more interpretable models tend to be less effective than complex, discriminative structures. Okutan et al. [115] integrate established code metrics with two newly introduced measures, the Number of Developers and the **Lack of Coding Quality (LOCQ)**, and use Bayesian Networks to model the causal relationships among metrics, as well as between metrics and defect proneness. Experiments conducted on the PROMISE data repository [93] highlight the prediction effectiveness of three metrics: LOC, LOCQ, and Response for Class, a CK metric.

A potential indication of the presence of software faults may also come from the change history of source code. In particular, the code age and the number of previous defects can be indicative to estimate the presence of new bugs [52]. This type of analysis better reflects a software model where changes are continuously applied to a code repository and where new defects are

potentially introduced with each new release. While the majority of early SDP contributions have approached the problem from a single-release perspective, a second category of works (the so-called changelog approaches) have concentrated their efforts around software history, considered a more determining factor than code metrics to estimate defect density. Graves et al. [52] divide software quality predictors into product measures (like code metrics) and process measures, which quantitatively describe the change history of a software project, advocating for the latter category (product measures are considered inconclusive from a correlation analysis). Generalized linear models are constructed from process measures to predict the defect count in the software repository of a telephone switching system. A weighted-time damp model is also introduced, where code changes are down-weighted based on age, showing the best performances overall.

Ostrand et al. [117] examine changes in large software systems and their relation to past faults to predict the number of defects in the next release. As in Reference [52], they adopt a Poisson generalized linear model from which the maximum likelihood estimates of the model parameters are used to interpret the relevance of the different metrics. The model is tested on the release cycle of an in-house inventory system with a wide range of new metrics at the file level, including programming language, edit flags, and age. According to the results, the top 20% files with the highest predicted fault count contained on average 83% of the later identified faults. A comparative analysis of the two sets of SDP metrics (code and change metrics) was conducted by Moser et al. [98] on the Eclipse project repository. The comparison is performed using three different Machine Learning approaches: Naïve Bayes, logistic regression, and decision trees. Change metrics used individually are shown to be more efficient than code metrics alone for detecting defective source files. Moreover, a combined approach slightly improves or has comparable results to a change metric approach. Giger et al. [50] also utilize a combination of source code and change metrics to tackle defect prediction at the method level by applying four different ML algorithms. Using an investigation methodology similar to Reference [98], results are again presented by the set of input metrics used with similar conclusions: change-metric approaches outperform code-metric approaches, while the advantage of a hybrid set of metrics is observable but limited.

In addition to introducing the benchmark SDP dataset AEEEM, D'Ambros et al. [35] compute several change-related metrics, such as the number of revisions, refactorings, and bug fixes per file, which are correlated with the number of future defects. Mostly based on the concepts of code entropy and churn, these metrics are then applied with generalized regression models to classify and rank files by probability defect. The proposed ranking approach can consider the necessary review effort of files as well. **Learning-to-Rank (LTR)** is also the focus of the work of Yang et al. [153], where a linear model is trained via **composite differential evolution (CoDE)** and input metrics are selected using the information gain criterion. According to the authors, maximizing the ranking performance directly, rather than relying on the predicted number of faults, provides benefits such as robustness for SDP models focusing on ranking. This claim is verified by comparing LTR approaches with traditional least-squares regression approaches based on bug count prediction, where LTR approaches are in fact predominantly more accurate for the ranking task, especially in projects with a high number of metrics and releases.

Cross-project transferability, i.e., the ability to learn a software defect model on source projects different from the target domain, represents the major application obstacle to some authors. Nam et al. [103] study the problem of transfer learning for software defects by applying an extended version of **Transfer Component Analysis (TCA)** to learn common latent factors between source and target projects. The extension (TCA+) improves cross-project prediction by selecting an appropriate normalization option for source and target. TCA+ features are then used as input of a logistic regression classifier to predict faulty module files. The approach is validated on two datasets (Relink [147] and the already mentioned AEEEM [35]): documents are represented by the different

Table 5. Summary of Described Methods for SDP, Categorized by Data Sources, SDP Task, and Target Code Unit

Paper(s)	Data Sources			SDP Typology		Target Code Unit			
	Code Metrics		AST data	Same-project	Cross-project	Function	Class	File	Module
	Static	Change-related							
[102]	•			•	•				•
[38]	•			•				•	•
[94]	•			•					•
[115]	•			•			•		
[42]	•			•		•			
[117]		•		•				•	
[50]	•	•		•		•			
[35, 98]	•	•		•				•	
[153]	•	•		•					•
[52]		•		•					•
[76]	•	•	•	•				•	
[103]	•	•		•	•			•	
[141]			•	•	•			•	

software metrics available for each dataset, and results are compared to other prediction methods using the F-measure (for cross-project prediction, 0.61 on Relink and 0.41 on AEEEM).

A critique of traditional code metrics is that they are too handcrafted and simplistic. An alternative is to directly parse the source code using **Abstract Syntax Trees (AST)**. Wang et al. [141] debate the ability of code metrics to model semantics and thus discriminate between code regions with the same structure and different semantics, arguing in favor of latent semantic representations. In their work, after parsing the code with ASTs, a **Deep Belief Network (DBN)** is trained to learn semantic features in an unsupervised fashion via stochastic first-order optimization. Then, Naïve Bayes, Decision Trees, and Logistic Regression models are used to predict faults from the latent representations in both within- and cross-project settings. Results obtained on Java projects of the PROMISE repository dataset are compared with TCA+ [103] and show a significant improvement in terms of F-score in the cross-project setting (0.568, +9.1% over [103]).

Li et al. [76] explore **Convolutional Neural Networks (CNN)** for SDP. During the parsing step, a subset of AST nodes corresponding to different types of semantic operations is extracted. These tokens are mapped to numerical features using embeddings and fed into a 1D convolutional architecture, which is used to learn intermediate representations of the input code, and later integrated with handcrafted features for the final prediction. Results obtained on a smaller subset of projects of the same PROMISE dataset are compared with a traditional logistic regression model and the DBN approach of Reference [141], exhibiting again a substantial F-score improvement (0.608, +0.084 over traditional methods and +0.065 over Reference [141] for within-project SDP).

4.1.2 Fault Injection. Fault injection is the deliberate introduction of faults into a target working system [5] to evaluate the level of fault tolerance reached. In traditional computer systems, this evaluation represents a validation of the reactive capabilities of a system off-the-shelf. In the specific case of AIOps and failure management, fault injection also allows one to evaluate the efficacy of externally deployed reactive mechanisms. Several works presented in later sections have

applied fault injection techniques for this purpose [25, 87, 110, 119, 124, 127]. Important definitions in fault injection are the set of injected faults or faultload (F); the set of activations of faults in the system (A); the set (R) of readouts from the system and the set (M) of measured values. F and A model the injection procedure from the input, R and M from the output [5].

As the concept of fault injection is disconnected from the specific type of system under test, its use is not restricted to a single applicability scenario. Fault injection can be applied at the hardware level, to emulate the appearance of faults in physical components, such as hard drives and CPUs, as it is infeasible and costly to induce real faults in this domain. At the software level, fault injections are used to understand the effect of bugs in the behavior of computer software or to model the causal dynamics of one or more software components in a shared-resource environment, such as an operating system and the executing processes. Software Fault Injection (SFI) is the automated and planned insertion of software faults (which in this case constitute the faultload) [105]. Software faults can be inserted directly or can be simulated to model the consequences of their injection. For instance, the injection of erroneous data in a program introduces a failure, representing an undesired behavior, but not a root-cause problem (i.e., a fault), therefore modeling the problem indirectly. Another common target for data corruption is function interfaces, such as APIs.

A concept connected to SFI is **Mutation Testing (MT)**, i.e., the introduction of small modifications in the source code to evaluate the quality of code coverage achieved by test cases. MT comes closer to mimicking faults compared to error injection, because it introduces software faults directly rather than their consequences (the errors). However, introducing random code changes, targeting variable assignments and function calls, can become inefficient if the procedure is not guided, due to the expensive cost of running the test case suite for each of the exponentially numerous mutated versions of the program. It is therefore fundamental to reduce the number of mutations to the minimum essential, without losing the discernment ability of mutation operations. Nowadays, many available tools try to identify code changes that can be representative of real software faults. While, on the one hand, MT operates to improve the quality and coverage of test cases, SFI, on the other hand, attempts to identify and emulate faults that cannot be discovered by test suites, because testing is performed in sandboxed scenarios, diverging considerably from real operation workloads. SFI covers this gap by stressing the operating conditions of a production environment while examining the consequences.

In a distributed computing environment, fault injection can also be applied at the cluster or datacenter scale, for example, to randomly terminate operations at the instance level and stress-test the resiliency of the distributed service under investigation. This is the case with software tools such as Chaos Monkey and Kong by Netflix [13] or Facebook's Project Storm [69]. Finally, fault injection techniques can be applied at the network level based on principles that are similar to the hardware and software level previously cited.

Nowadays, approaches performing fault injection are mostly out of the realm of AI, as fault injection predominantly consists of switching off mechanisms and fault insertion procedures, where the problem is purely algorithmic and the necessity of intelligent agents is unjustified. The few available contributions of AI to the field are concentrated around the implementation selection policies for the faultload, where the number of applied faults needs to be reduced for computation reasons.

In the context of MT, Siami Namin et al. [128] use a regression-based model to estimate the efficacy of a subset of mutation operators in predicting the **mutation adequacy ratio (AM)**, a measure of test coverage in MT. In particular, for each mutation operator, they compute the individual AM and use it as a regression feature to estimate the overall AM of the test suite. They argue that a minimal subset of mutators able to predict the AM measure will also represent the ideal, reduced subset of operators to apply for mutation testing. They also apply other statistical analysis

techniques, such as clustering and correlation, to identify groups of similar variables and eliminate the one with the highest computation cost from the selection domain. Using this technique, the set of code mutants is reduced to 8% of the initial size, drastically reducing the estimation time for the AM measure, which is in turn necessary to estimate code failure coverage.

Motivated by the strong presence of unrepresentative faults that are obtained with state-of-the-art SFI approaches, Natella et al. [105] use two Machine Learning approaches to improve the faultload set. They apply decision trees for classifying components into higher and lower fault risk (MR and LR, Most and Least Representative, respectively) for consequent fault selection. Moreover, starting from the observation that low fan-in and fan-out components have faults that are more difficult to detect, the authors apply an unsupervised learning approach using the code metrics of software components (see also Section 4.1.1). Specifically, k-means clustering is used to separate target components into two clusters based on their interaction. Faults are injected into components of the cluster with the lowest fan-in and fan-out. The approach is tested on different software components (MySQL, RTEMS, PostgreSQL) at different granularities and can reduce faultload size (−22% to −69%) while increasing fault representativeness (+4% to +26%).

4.1.3 Software Aging and Rejuvenation. Software aging describes the process of accumulation of errors during the execution of a program that eventually results in terminal failures, such as hangs, crashes, or heavy performance degradations [49]. Known causes of software aging include memory leaks and bloats, unreleased file locks, data fragmentation, and numerical error accumulation [21].

Several Machine Learning techniques have been applied to predict the exhaustion of resources preemptively. Garg et al. [49] estimate time-to-exhaustion of system resources, such as free memory, file and process table sizes, and used swap space, using instrumentation tools available under the UNIX operating system. Resource measurements are collected on several operating workstations to construct time series, which are annotated with the failure cause in case of outage. Trends are detected and exhaustion time is quantified using regression techniques and seasonal testing.

In addition to the time-based measurement of resources, Vaidyanathan et al. [136] investigate the effect of software aging due to the current system workload. A semi-Markov reward model is fitted from the available workload and resource data under UNIX, where the model states represent different workload scenarios, and the association to a specific state is identified employing Hartigan's k-means clustering. Time-to-exhaustion of memory and swap space is estimated with a non-parametric regression technique for each workload state separately. Alonso et al. [4] consider the case of non-linear and piecewise linear resource consumption adopting an ensemble of linear regression models, selected using a decision tree on the same input features of the regression model (a combined set of hardware and software host metrics). Decision trees are chosen because they offer a trade-off between accuracy and interpretability. The authors propose to use their interpretability for root-cause diagnosis as well (Section 4.4.2).

Software aging can be contrasted with software rejuvenation [57], a corrective measure where the execution of a piece of software is temporarily suspended to clean its internal state. Software rejuvenation can be performed at the software and OS level. Common cleaning operations include garbage collection, flushing kernel tables, re-initializing internal data structures [136]. According to Castelli et al. [21], software rejuvenation fits naturally inside a cluster environment (but similar considerations are true for any distributed computation system, e.g., cloud), because of the already present node failover mechanisms and the possibility to schedule rejuvenation and aging prediction routines. To this end, authors distinguish between periodic (or synchronous) and prediction-based (or asynchronous) rejuvenation policies, where the latter requires a prediction model for future software failures. The failure prediction model can belong to (but is not limited to) the category of resource exhaustion predictors described above.

AI has been used in the past to suggest efficient rejuvenation scheduling policies. The same Castelli et al. [21] apply stochastic reward nets to model service downtimes and draw conclusions on the efficiency of rejuvenation policies. Their analysis considers both time-based and prediction-based policies. For the latter, they resort to resource exhaustion prediction algorithms to estimate future failure periods, ideal for refreshing the internal state and examine the expected downtime as a function of the prediction model accuracy (or prediction coverage). Both periodic and prediction-based policies are shown to be able to reduce downtime significantly, with prediction-based methods having a larger improvement overall (−60% downtime at 90% coverage versus −25% with optimal time-based), while high-frequency periodic policies can better tolerate simultaneous failures (−95% versus −85% of prediction-based methods).

In Reference [137], Vaidyanathan et al. build upon their workload-inclusive Markov prediction approach [136] to derive optimal periodic rejuvenation policies. To capture the manifestation of rejuvenation and failure behavior, they develop a comprehensive transition-based model between three states (working, failure, and rejuvenation). The model parameters (such as the period of rejuvenation) are optimized to reach different objectives (e.g., maximum steady-state availability or minimum downtime cost). A similar strategy allows one to improve the preventive strategies of a distributed system during its design stage.

4.1.4 Checkpointing. A concept linked to software rejuvenation is checkpointing, i.e., the continuous and preemptive process of saving the system state before the occurrence of a failure. Similar to software rejuvenation, checkpointing tolerates failures by occasionally interrupting the execution of a program to take precautionary actions. Different from software rejuvenation, during checkpointing the interruption period is used to save the internal state of the system to persistent storage. In case a fatal failure occurs, the created checkpoint file can be used to resume the program and reduce failure overhead. Checkpointing techniques are common in distributed computing, Checkpointing is also frequently used in large-scale computing systems, although with the increase of scale the higher error rates make it less practical [62, 95]. To this end, the concept of multi-level checkpointing was introduced: checkpoints are created at different component levels, and based on the component resiliency and the permissible time overhead, a different checkpointing strategy is selected for each level.

As in the case of software rejuvenation, AI is used to model a faulty execution workload under different checkpointing strategies and select the most suitable strategy according to different objectives. Again, checkpointing can be static or dynamic depending on the scheduling of checkpoints. All the papers investigated operate within the framework of Markov processes [62, 95, 114]. Okamura et al. [114] develop a dynamic checkpointing scheme for single-process applications based on reinforcement learning. In particular, Q-learning is used to obtain the optimal checkpointing by studying the asymptotic behavior of policies. The approach is especially useful in scenarios with unknown failure characteristics.

Moody et al. [95] describe a multi-level checkpointing system for large HPC systems. To estimate execution times under different configurations, the multi-level system is expressed as a joint Stochastic Markov Model with the definition of computation and recovery states and the introduction of assumptions such as the cost of checkpointing and the failure rate of the system. The final checkpointing schemes are claimed to offer high recovery efficiencies (85%) at a 50× increased failure rate.

Jangjaimon et al. [62] present an incremental checkpointing strategy for multi-threaded applications in a cloud environment. Their system makes use of an Adjusted Markov Model to predict turnaround time in the execution of programs, incorporating both the effect of hardware failures and potential resource revocation events. The model also takes into account monetary aspects

starting from cost and unavailability assumptions. According to their experimental results, an adaptive multi-level checkpointing scheme is beneficial for cloud paradigms with resource revocation, allowing reduced checkpoint sizes, shorter execution times (up to −25%), and lower down costs (up to −20%).

4.2 Online Failure Prediction

The fundamental idea behind failure avoidance is the anticipation of errors. The prediction of failures is therefore an essential action for proactive management of failures. The previous section has illustrated how estimating the distribution of future failures is valuable to operate preemptively against them. In all previous instances of prediction, however, forecasting was strictly connected to the specific prevention task undertaken and also largely dependent on the available preventive actions (e.g., software rejuvenation). Moreover, except resource exhaustion prediction and dynamic checkpointing, all previous instances of prediction operate using assumptions derived in an offline setting, estimating the probability of failures occurring in absolute rather than estimating the time to the next failure. A prediction of failures on-the-fly allows one to be aware of future failures but also to know in advance the remaining useful to recovery, vital to timely deploy recovery and failover mechanisms.

This section discusses this second category of proactive approaches, which are specialized in prediction of computer system failures in this online fashion. Online failure prediction identifies future runtime errors by assessing the current state of the system [122]. How far in time these errors can be foreseen depends on the lead time of the predictor, i.e., the time between the prediction and the instant when the failure occurs; the validity of the prediction information also depends on the prediction time, which is the length of the time window where the failure may occur. To these prediction quality measures, we also add the warning time, a metric describing the time required to perform inference and signal a future failure. Naturally, a prediction is useful only if it can warn operators with sufficient advance before the failure occurs (i.e., $t_{warn} < t_{lead}$). From a functional perspective, an online failure predictor is comparable to a detector and is, as such, evaluated in terms of accuracy, precision, recall and the other associated measures. We discuss online failure prediction approaches into two subcategories: hardware failure prediction and system failure prediction.

4.2.1 Hardware Failure Prediction. In large-scale computing infrastructures, hardware reliability represents one of the most relevant practices to achieve service availability goals in distributed services. Due to the magnifying effect of the large numbers involved and the commercial necessity to deploy commodity components in datacenters, hardware represents the most vulnerable aspect from a failure perspective. According to Vishawanath et al. [138], 8 servers of 100 are expected to encounter at least one hardware error per year of operation. Moreover, the machines affected by errors are likely to require more than one repair per year (with an average of two repairs), showing a successive correlation pattern between failures. According to the same work, hardware repairs for a 100k-scale datacenter can amount to millions of dollars. It is therefore crucial from the industrial perspective to investigate which factors influence the appearance of hardware faults for the end goal of improving design choices and deploying failure predictors.

Hard drives are the most replaced components inside large cloud computing systems (78%) and one of the dominant reasons for server failure [138]. Machines with a higher number of disks are also more prone to experience additional faults in a fixed time period. This fact has led hard-drive manufacturers to adopt common self-monitoring technologies (such as **Self-monitoring Analysis and Reporting Technology (SMART)**) in their storage products since 1994 [100]. Therefore,

it should not be surprising that hard drive failure prediction represents the most investigated target for failure prediction.

Several studies have investigated the failure characteristics of hard drives to identify common patterns. For instance, Pinheiro et al. [118] conducted a large-scale study on over a hundred thousand disk drives used in production by Google and varying in storage size, speed, and manufacturer. The results of this study could not identify any consistent correlation between failure rate and high temperature or high utilization levels. However, some SMART features were shown to correlate well with a higher failure rate. However, SMART metrics were also shown to be likely insufficient for single-disk predictions, as the majority of failed drives did not manifest any SMART error signal before faults. A predictor based solely on SMART attributes is therefore likely to have good specificity but low sensitivity unless additional features are introduced. SMART data is still considered useful to evaluate reliability and risk trends inside a disk drive population. In the work of Vishwanath et al. [138], different other attributes are investigated from a population of disks in Microsoft servers. By constructing decision trees for failure prediction two discrimination factors are identified, datacenter and manufacturer, while the type of workload, the position in rack, the server age, and the configuration are all found to be of no significance for predicting future drive faults.

Hamerly et al. [54] present two different Bayesian learning approaches for disk failure prediction from SMART attributes collected on a real-world dataset of 1936 drives. The first method works as an unsupervised anomaly detector, where a mixture of Naïve Bayes discrete submodels is trained to estimate the posterior probability of observation, and a hard-set threshold is used to separate anomalous events from normal behavior. The second method consists of a supervised Naïve Bayes classifier with binned continuous variables. The approach achieves a recall of 0.33 with a false-positive rate of 0.0067. Both methods operate by predicting a failure at the snapshot level, so the temporal evolution is not taken under consideration.

Different from previous works [54, 101], Zhao et al. [164] treat SMART data as a time series, arguing for the importance of temporal information. Their approach employs **Hidden Markov and Semi-Markov Models (HMM/HSMM)** to estimate the sequence of likely events from the disk metric observations, which are obtained from a dataset of approximately 300 disks (where approximately two-thirds were healthy). One model is trained from healthy disk sequences, one from faulty disk sequences. At test time, the two models are used the estimate the sequence log-likelihood and the corresponding class is selected based on the highest score. By combining the HMM approach with an SVM, they claim to obtain a recall of 0.52 at 0 false alarm rate.

Murray et al. [101] test the applicability of several Machine Learning methods using a sliding window approach, where the last n samples constitute the observation for predicting an imminent failure. Naïve Bayes, kernel SVMs, and Naïve Bayes Expectation-Maximization as already proposed by Reference [54] are implemented and compared. Features are selected from SMART data using statistical relevance tests. SVMs achieve their highest performance with a 50.6% recall and 100% precision. The approach was tested on a dataset composed of 369 drives (with an approximate 50/50 split), which is then also used in a work by Wang et al. [143] where an online, similarity-based detection algorithm is presented. Relevant SMART features are selected via **Minimum Redundancy Maximum Relevance (mRMR)**, then the input data is projected into a Mahalanobis space constructed from the healthy disk population so that faulty disks deviate more from the distribution. Faulty disks are again recognized with a sliding window approach so that when the mean deviation inside a window appears anomalous an alarm is raised. An anomaly is identified with four different statistical tests. This approach improves, at 0 false-positive rate, the detection rate up to 67%, on the same dataset with less computational effort. Moreover, it is shown how for 56% of the faulty cases it was possible to intervene with an advance of at least

20 h before the failure. Comparable results are obtained by Zhu et al. [167], where multilayer perceptron and SVM models are constructed and trained on an in-house (Baidu) dataset of SMART data comprising 23,395 drives (433 faulty). Assuming a 12-h recovery window, their SVM model obtains a failure detection rate of 68.5% with a 0.03% false alarm rate. The neural network method achieves far higher detection rates (94.62–100%), at the expense of a higher false alarm rate as well (0.48–2.26%) and it is therefore indicated for monitoring with the highest reliability requirements. The same SMART dataset is used by Xu et al. in a paper [149] that introduces **Recurrent Neural Networks (RNN)** to hard drive failure prediction. Similar to Reference [164], the method can analyze sequences directly and to model the long-term relationships takes advantage of the temporal dimension of the problem. Differently from previous approaches that apply binary classification, the model is trained to predict the health status of the disk, providing additional information on the remaining useful life of disks. On the failure prediction task, however, the approach still outperforms the other evaluated models in terms of detection rate (96.08–97.78%) and false alarm rate (0.004–0.03%).

In a work [78] related to Reference [167], two new evaluation metrics (migration and mismigration rate) are introduced to measure the efficiency of data migration concerning forecasted faults. RNNs and **Gradient-boosted Decision Trees (GBRT)** are implemented (regression trees originally proposed in Reference [77]) to accomplish three tasks: predict faulty disks, as in the usual setting; measure the rate of wrongful and missed migrations performed using the information of the classifier; estimate the residual life of the disks, by predicting the risk level of each disk (1 to 5 for faulty disks, 6 for healthy ones). For residual life prediction, results are compared using the newly defined metrics at variable migration rate. The RNN approach shows a higher faulty-level prediction accuracy (27.02–39.90%) and the GBRT model better protection from data loss with a higher successful migration rate (84.91–87.54%).

Mahdisoltani et al. [89] tackle failure prediction in storage media at the sector level. Their method employs a few SMART features as target prediction variables rather than explanatory variables. They experiment both with HDD and SSD data, with five different Machine Learning approaches. For HDD data the analysis illustrates detection rates of sector errors similar to the ones of traditional disk failure prediction (70–90% at 2% false-positive rate). In SSDs, where Random forests obtain the best comparative results, the prediction is not as promising (50–60% detection rate at 2%). SSD failure characterization is also the focus of Narayanan et al.'s work [104], a large-scale study conducted on 2.5 years of production data and covering over 500,000 solid-state disks from different Microsoft datacenters. SSD failures are correlated to datacenter, workload, and device-level features via statistical learning. The best and only reported prediction model (random forest) obtains a precision of 87% and a recall of 71%. Features directly representing underlying problems (such as count of data errors and reallocation sectors), number of NAND writes and workload factors are reported as the most important discriminators of failures. According to the same authors, the classification rules obtained by the analysis enable the identification of likely-to-fail devices with sufficient advance to take preventive actions, especially in the case of issues heavily dependent on the workload.

All previous approaches are used in an online setting after an initial setup or training step is performed offline. This poses the problem of how to integrate additional data, especially in those cases where the scarcity of positive training examples imposes an online learning approach, able to update the characteristics of faulty disks as soon as new failures appear. To this end, Xiao et al. [148] propose the use of Online Random Forests, a model able to evolve and behave adaptively to the change in the data distribution via online labeling. The approach is tested on a dataset covering over 10,000 disks, where the results show how the algorithm can increase performance over 20 months, reaching detection rates of 93% and more with reasonably low alarm rates

(0.73–0.76% in the offline setting). The prediction performances are comparable to a Random Forest and outperform the other approaches tested (SVMs and decision trees).

Although most of the scientific interest is concentrated around disk failures, a minor group of contributions, focusing on failure prediction for other components, is present. Ma et al. [88] elevate the discussion on storage reliability to the level of RAID groups. They model the failure probability of a vulnerable RAID group using a Naïve Bayes assumption, where the failure of whole disks inside the group is independent of the others and is estimated from metrics such as reallocated sector counts. A statistical model is built from data collected on 5,000 RAID groups. Results show how the present in-place mechanisms were able to prevent a vast majority of consecutive failures (98% for triple failures).

Costa et al. [33] investigate the occurrence of main-memory errors for HPC applications. They propose and evaluate methods for temporal and spatial correlation among memory failures. Temporal correlation analyses logs to estimate the prior rate of errors that, integrated with timing information, allows one to estimate the remaining number of errors at the job level. Spatial correlation measures the probability to experience a failure after observing errors in adjacent bits. In the same paper, a memory migration approach, based on the available information at runtime, is presented and evaluated. The results show how 63% of memory-induced failures could be avoided thanks to the prediction and migration mechanisms deployed.

Davis et al. [37] present FailureSim, a Cloudsim-based [51] simulator for status assessment of hardware in cloud datacenters using deep learning. Multilayer Perceptrons and RNNs are used to assess 13 different host failing states, each associated with a specific component (CPU, memory, I/O, etc.). The system, tested by assigning a variable workload to a scalable number of VMs, can identify 50% of failing hosts accurately, and predict 89% of future fails before their occurrence.

Zhang et al. [161] deal with the problems of failure prediction and diagnosis in network switches. Their method, based on system log history, proposes to extract templates from logs and correlate them with faulty behavior. Their extraction method and similar approaches are compared on the extraction tasks, and the templates there obtained are used to train a Hidden semi-Markov Model. The proposed model, which shows the best performance figures, achieves 32% precision and 95% recall, demonstrating high sensitivity to clear fault-signaling messages, but low precision (and thus high false alarm rate) overall.

Zheng et al. [165] propose a method based on RNN to estimate the Remaining Useful Life of system components. They argue that the temporal nature of sensor data justifies the use of **long short-term memory (LSTM)**-based RNNs, due to their ability to model long term-dependencies. They experiment with three RUL public datasets and compare their method with other Machine Learning approaches in terms of **Root Mean-squared Error (RMSE)**, showing how their approach obtains the best RUL prediction performance (RMSE = 2.80, 54.47% improvement over CNNs).

4.2.2 System Failure Prediction. Instead of investigating the occurrence of physical component failures, future system availability can also be estimated through symptomatic evidence and dependency modeling assumptions at the software level. Past approaches for system failure prediction are mostly based on the observation of logs, which constitute the most frequent data source, KPIs and hardware metrics, which are typically used in shorter prediction windows and are more frequently associated with the failure detection problem as well. System failure prediction are applied on different abstraction levels depending on the target software component under investigation (job, task, container, VM, or node).

Some early contributions are associated with failure prediction in the IBM supercomputer BlueGene project. Liang et al. [81], for instance, apply different classification algorithms for BlueGene,

using real event logs to predict fatal failures inside fixed time windows of observation. Input features are parsed from structured logs based on the severity, total count, and distribution of log events inside an observation window. Features obtained from the previous window are used to predict failure in the next one. In the evaluation phase, four classification algorithms (SVM, RIPPER, BMNN, *k*-NN) are compared using precision, recall, and F-score. The size of the prediction window appears to have a large impact on the final detection performance. From the experiments, the ideal window size is estimated between 6 and 12 h to balance utility effectiveness and accuracy. The Bi-modal nearest neighbor approach proposed in the paper provides the best results overall, leading to an F-measure of 70% (and 50%) with 12-h (and 6-h) prediction windows, respectively.

Cohen et al. [31] investigate an approach based on **Tree-augmented Bayesian Networks (TANs)** to associate observed variables with abstract service states, forecast and detect SLO violations and failures in three-tiered Web services. The system is based on the observation of system metrics, such CPU time, disk reads, swap space, and KPI-related measures, like the number of served requests, all interdependently modeled (in addition to the dependent variable, the predicted state). The optimal graph structure, including the ideal subset of input metrics, is selected utilizing a greedy strategy. The approach, originally developed for detection, can also be used to diagnose failures thanks to the interpretability properties of TANs (see Section 4.4.2). Two separate experiments are used for validation, a multi-tiered Java server application, and an Apache server testbed, both injected with synthetic workloads varying in connections, request rate, and type. The results for forecasting, applied 1 or 5 min in advance of failures, show high detection rates (83–93%) and false-alarm rates varying between 9.1% and 24% depending on the experiments.

Salfner et al. [123] train HSMM for online prediction of failures in event sequences collected from error logs. One HSMM is trained on failing sequences, a second one on non-failing sequences. Then, the sequence likelihood of the models determines which of the two scenarios is more likely to occur. The approach is evaluated on logs of a commercial telecommunication system and compared with other prediction techniques, achieving an F-measure of 0.7419 (precision 0.852, recall 0.657). The observed false-positive rate is 0.0145.

Chalermarrewong et al. [23] propose a system availability prediction framework for datacenters, based on autoregressive models and fault-tree analysis. Their autoregressive integrated moving average (ARIMA) model works on time series of workload and hardware metrics, where thresholds are fixed to detect component-level symptoms (such as high CPU temperature, bad disk sectors, memory exhaustion), which also constitute the leaves of the fault tree. Using these symptoms and the tree structure, a model of dependencies in combinational logic determines the availability state deterministically. An advantage of these approaches that it implicitly provides granular information for diagnosis. The experimental results show high prediction accuracy (97%) but a high rate of false alarms as well (precision is 53%).

The HORA prediction system [119] adopts a holistic approach, where architectural knowledge is used with online time-series data to predict QoS violations and service failures in distributed software systems. Component dependency and failure propagation models, in the form of Bayesian Networks, are used to associate component failures, predicted from system metrics with autoregressive predictors, to system-wide problems. Tested on a microservice-based application, the new approach is compared to a monolithic approach (not using architectural knowledge) by injecting memory leaks, node crashes, and system overloads during execution. The HORA approach achieves a higher recall (83.3% versus 69.2%) and AUCROC (0.920 versus 0.837, +9.9%) compared to the monolithic approach. The proposed approach is also more viable when a high false alarm rate ($\geq$10%) is acceptable, while the monolithic approach is more adequate for predictions in the 0–10% false alarm range.

Fronza et al. [44] describe a failure prediction approach based on text-analysis of system logs and SVMs. First, log files are parsed and encoded using a technique called Random Indexing, where for each operation inside a log line, index vectors are assigned to obtain a latent representation of the text. Then, by scanning through the log corpus, context vectors for each operation are computed, so that co-occurrence of operations can be taken into account. Finally, sequences of operations are represented as the weighted sum of context vectors corresponding to the operations encoded inside. These representations of sequences constitute the input of a weighted SVM, trained with the objective of minimizing false negatives. Results show how a weighted approach can compensate for a decrease in specificity (always greater than 80%) to improve recall (50%).

Similarly, in Reference [160] RNNs are applied for failure prediction from logs. The approach is composed of a clustering algorithm used to group similar logs, a pattern matcher used to identify common templates inside similar logs, a log feature extractor based on natural language processing, and a sequential neural network architecture based on LSTM cells, used to predict the failure status within the predictive period. The use of LSTMs is also motivated by their sequential modeling abilities and the rarity of failures. The approach is compared, using real data collected from two enterprise server clusters, with other Machine Learning methods (SVM, logistic regression, random forest) in terms of precision-recall AUC. At 70% precision, the LSTM method shows the highest failure sensitivity with a recall value of 0.909. LSTMs are able to detect signs of failures earlier than other methods (73 min on average).

LSTM networks are also used by Islam et al. in Reference [61], a large characterization study of job failures conducted on a workload trace dataset by Google. Failures are predicted at the job and task level, where a job is a collection of tasks and each task is a single-machine command or program. Failures are predicted from resource usage measures, performance data, and task information (completion status, user/node/job attributes). For task failure prediction, the final F1-score of the system is 0.87 (precision = 0.89, recall = 0.85, FPR = 0.11). For job failure prediction, the approach achieves a F1-score of 0.81 (precision = 0.80, recall = 0.83, FPR = 0.20). The prediction algorithm can be used to reduce resource waste (down by 12–20% in the experiments) by re-submission of likely-to-fail jobs and tasks.

4.3 Failure Detection

In this section, we start discussing reactive failure management methods, which operate to limit the consequences of failures after they have occurred. They are motivated by the fact that, even with the most advanced prevention and prediction techniques, the occurrence rate of failures can never be fully reduced to 0. Some reactive approaches, for example in root-cause analysis or anomaly detection, can also be used to get a better understanding of how failures are causally related and propagate over time, or to comprehend with which temporal characteristics (variance, burst frequency, periodicity, seasonality) failures are associated.

The detection of failures via monitoring operations can be a complex and tedious task for human operators. Chen et al. [26] report how in the administration of a commercial website, 75% of the recovery time was spent on average for detection. The automatic discovery of performance problems and errors allows operators to dedicate less time identifying service-related problems while providing insights on which failures must be prioritized in the diagnosis step based on the frequency observed in the detection phase. Automated failure detection is based on a variety of monitoring tools, ranging from simple print statements (which constitute the fundamental unit of system logs) to more complex instrumentation techniques or entire frameworks. We divide our discussion of failure detection in anomaly detection, Internet traffic classification, and log enhancement techniques.

4.3.1 Anomaly Detection. According to our quantitative results [113], failure detection is treated as an anomaly detection problem in the large majority of contributions related to IT system management. Anomaly detection is a multidisciplinary task that deals with finding patterns in data that do not conform to the expected behavior [24]. The interest in anomaly detection is motivated by the possibility to obtain actionable information to deal with diseases, frauds, cyber-attacks, system outages, and faults, depending on the target domain. Anomaly detection operates by deriving a model of normal behavior, and by testing new observations against this model. Anomaly Detection is applied in the AIOps context under the assumption that failures generate irregular behavior in IT systems (errors) across a set of measurable entities (or symptoms), such as KPIs, metrics, logs, or execution traces. Anomaly detection is also used to detect cyber-attacks, congestion, and suboptimal resource utilization, which can all be causes of future failures. Because obtaining labeled examples is time-consuming, anomaly detection systems typically rely on unsupervised learning. Three are the most prominent techniques used in such context: clustering [11, 127], dimensionality reduction [72, 151] and neural network autoencoders [7, 131, 150, 159]. One of the earliest examples of unsupervised learning for system workload characterization is Magpie [11], a behavioral modeling toolchain for distributed systems. Magpie's instrumentation is based on fine-grained, in-kernel event logging tools available under Windows, able to measure resource consumption and execution times accurately. The events registered with this tool are used to reconstruct requests and apply behavioral clustering to obtain a small set of unique request types, each associated with a specific workload model. In the paper, behavioral clustering allows one to model realistic workloads more precisely than by grouping requests by URL. A realistic workload model is beneficial to identify anomalous resource consumption, besides enhancing capacity planning and performance testing.

At the network level, Lakhina et al. [72] monitor network links via SNMP data to detect and diagnose anomalies in network traffic. The apply PCA on link flow measurements collected over time to separate traffic in normal and anomalous subspaces. They classify principal components in normal and abnormal (setting a threshold on the explained variance), then identify anomalies by reconstructing the new observations using abnormal components. If the reconstruction error exceeds a predefined threshold, then the new data point is considered anomalous. Their approach is validated using synthetic and real volume anomalies, the latter detected with traditional class forecasting algorithms. In a follow-up work [73], a similar approach based on traffic feature distribution entropy is proposed. The paper shows how the same subspace method can exploit packet information, such as source and target destination address or port, to detect security threats and service outages, as well as to provide preliminary diagnostic information.

Several approaches apply unsupervised learning on univariate time series constructed from KPI and metric observations [127, 150]. Sharma et al. [127] propose CloudPD, an end-to-end failure tolerance system performing failure detection, diagnosis, and classification in virtualized cloud environments (see also Section 4.4.2). The paper proposes to collect a variety of measures at the VM and physical machine level, including resource utilization, operating system variables, and application performance metrics. The paper then proposes three different unsupervised Machine Learning approaches for anomaly detection: ***k*-nearest neighbors (*k*-NN)**, HMMs, and k-means clustering. The k-NN detection approach is evaluated on three different benchmarks, where it beats four proposed baselines with an average higher recall (83–87%) and a lower false alarm rate (12–17%). Donut [150] performs anomaly detection on seasonal KPI time series using deep **Variational Autoencoders (VAE)** and window sampling. The approach is compared on three different datasets of 18 KPIs to Opprentice [86] and a baseline VAE performance in terms of F-score (ranging from 0.75 to 0.9), AUC (0.7–0.9) and average alert delay (4 to 12 min). One of the conclusions of the

authors is that autoencoder approaches require abnormal samples in addition to normal behavior samples.

The most recent time-series approaches focus on multivariate anomaly detection using autoencoders [7, 131, 159]. Two advantages of a multi-dimensional analysis are the ability to model the inter-correlations between different metrics/components, and to provide interpretable results for root-cause analysis, by attributing the detected anomaly to a specific metric or component. MSCRED [159] uses a convolutional-recurrent (ConvLSTM) autoencoder architecture to capture temporal patterns at different scales and thus identify abnormal time steps. In the evaluation study, MSCRED outperforms the baselines (SVM, ARMA, GMM, CNN) with an F1-score of 0.82–0.89, while being able to identify anomalies on scales $w = 10, 30, 60$ s. By computing the reconstruction errors on individual time sequences, MSCRED can also identify root causes more effectively (top-3 recall = 0.75–0.80). OmnyAnomaly [131] proposes a recurrent VAE model using stochastic variable connection and planar normalizing flow. As for MSCRED, the reconstruction error can be used to interpret anomalies and connect back to individual time series. OmniAnomaly achieves a detection F1-score of 0.8599 on three real datasets, including **server metric data** (**SMD** [108]) collected from a large-scale Internet company. SMD is also used for evaluation in USAD [7], which applies adversarial-based techniques to the autoencoder architecture for faster and more stable training. In USAD, the anomaly score is parametrized so that the sensitivity can be adjusted rapidly and on multiple levels for real industrial applications. On five real datasets, USAD achieves results comparable to OmniAnomaly (F1 = 0.791 for both), while reducing training time by 547 times on average.

Although in the majority of cases real labeled samples are not available, it may be possible to obtain behavioral set of rules or an execution model for the system. The model information becomes also very relevant when failures need to be diagnosed (see Section 4.4.2). With the use of a behavioral model, different pattern-matching approaches can be developed. For example, having access to a **probabilistic context-free grammar (PCFG)**, which defines the likelihood of event chains to occur in sequential input, allows an anomaly detection algorithm to single out unlikely structures and detect anomalies. Magpie [11] already proposes the use of PCFGs to model event sequences and detect anomalous requests. A similar approach is used in Reference [26] to detect failures by searching for anomalies in execution paths (i.e., traces). This path information is especially valuable to localize and diagnose failures (see also discussion of Reference [26] in Section 4.4.2).

More often, a behavioral model of the system is inferred from the past execution history, often expressed in the form of logs. An example of such a model is constituted by **Finite State Machines (FSM)** [12, 45]. In a work by Fu et al. [45], log entries are first mapped to their corresponding template version, called log key. Similar log entries, used to identify line templates, are grouped via clustering. Then, FSMs are learned to model program workflows from log evidence. The model so obtained can be later used to verify the correct execution of programs and detect software problems. The entire approach pipeline is tested on log files from two distributed computing frameworks, achieving 95% accuracy on log key extraction during the detection of different types of failures manually inserted. Beschastnikh et al. adopt a similar approach with their system CSight [12]. As the main goal of failure detection is to enable root-cause analysis, the authors also performed a user study on the usefulness of FSM diagrams compared to other graphical debugging tools for identifying execution errors. FSM diagrams enabled study subjects to identify problems more often (+11%) compared to subjects who were shown 6/8 time-space diagrams, considered less understandable by them. In Reference [29], a causal model of request executions is constructed from the available component-level trace logs. The model, called The Mystery Machine, identifies different types of predefined causal relationships between components via iterative refinement,

Table 6. Summary of Described Anomaly Detection Methods, Categorized by Data Sources and Algorithm, with Corresponding Advantages and Disadvantages

Paper(s)	Data Sources					Algorithm	Advantages/Disadvantages
	KPIs & Metrics		Logs	Traces	Others		
	Univariate	Multivariate					
[31]	•					TAN	interpretability, applicability to other tasks/requires supervision
[86]	•					Random Forest	high accuracy, online re-training/ requires supervised data
[110]	•					Markov Chain	unsupervised, online learning/ black-box monitoring
[127, 150]	•					Unsupervised Learning	general purpose, no labels required/ univariate analysis only
[7, 131, 159]		•				Autoencoder	multi-dimensional analysis, no labels required/training step
[12, 45]			•			FSM	enables RCA/requires template discovery step
[18, 41, 92, 162]			•			RNN	robust, accurate, unsupervised/ requires parsing, training step
[151]			•		•	PCA	simple, unsupervised method for free-text data/requires parsing, low interpretability
[11, 26]				•		Clustering/ PCFG	enables multiple AIOps tasks/ requires end-to-end tracing
[29]				•		Causal Inference	interpretability, enables RCA/ applicable in sequential systems only
[87]				•		SVM	high-level, large applicability/ requires supervised data, tracing
[72, 73]					•	PCA/Feature Distribution	specific/exclusive for network-traffic anomaly detection

supported by the past trace history. According to the authors, The Mystery Machine can be used to conduct an anomaly analysis of service requests based on segment features. In particular, it can used to aggregate similar requests by structure and latency via comparison of the segment structure of anomalous requests. Thanks to this inspection, it was possible to identify unnecessary debugging components returned occasionally in user requests. In a work by Xu et al. [151], text analysis and information retrieval techniques are applied on console logs and source code to find abnormal patterns during the operation of large-scale datacenters. In particular, state variables and object identifiers are extracted automatically from parsed logs, and their frequency in different documents (expressed in term inverse-document frequency, TF-IDF) is analyzed with PCA to detect anomalies with a threshold-based rule on the reconstruction error, similar to Reference [72]. Anomalies detected with the PCA approach are also used to train supervised decision trees, useful for operators to interpret anomalies and accelerate the identification of problems.

In recent years, RNN have been applied for log-based anomaly detection [18, 41, 92, 162]. Du et al.'s DeepLog [41] uses RNNs based on LSTM layers to learn patterns from logs and detect anomalies. As in Fu et al. [45], log entries are initially mapped to corresponding log keys. Then, DeepLog predicts the probability distribution of the next log key from the observation of previous

log keys, similar to a natural language model that predicts the next token from the observation of the rest of the sentence. A log key is considered abnormal if it does not appear in the top k keys ranked by probability. The paper also proposes an online learning strategy based on user feedback. The approach is validated on self-generated datasets from HDFS and OpenStack, where it is compared to other methods for the same task. At 100% recall, DeepLog dramatically reduces the false-alarm rate (from 38.2–40.1% to 1.1–1.7%) using only a small fraction of data for training (1–10%). Brown et al. [18] also work on log-based anomaly detection, presenting an word-token RNN language model for raw logs based on five attention mechanisms (reaching AUCROC $\geq$ 0.963). The paper also proposes to analyze attention weights to provide statistical insights, understand global behavior and improve decision making. Although their work is tested for intrusion detection tasks, it can be used for other detection tasks, e.g., detect failures at the request and server level. The same authors also recognize its potential to predict hardware failures online. LogAnomaly [92], while adopting the language model approach, proposes the use of template embeddings (template2vec) to better extract semantic information, by which similar log keys can be matched and merged automatically, without requiring human feedback. Thanks to template vectors, OmniAnomaly can also learn quantitative patterns in addition to sequential patterns. All these improvements enable to further reduce false alarm rate compared to DeepLog (F1 = 0.8632). LogRobust [162] deals with the problem of log instability, caused by the changing log statements over time and noise in log processing. To address this, LogRobust also employs LSTM layers, semantic vectors and attention, but differently from language model approaches, it directly predicts anomalies by computing an anomaly score. The approach is validated on industrial and synthetic HDFS data, with an average F1-score of 0.81 (+0.29) on unstable log data.

While unsupervised learning approaches are generally more applicable, supervised learning approaches for anomaly detection are viable in scenarios where labels are available for both normal and anomalous samples. Moreover, all types of anomalies need to be known beforehand and sufficient samples for each type must are necessary. Lo et al. [87] investigate the detection of failures in software as a classification problem. Their approach mines sets of discriminative features from execution traces and use them to train a SVM for classifying software behavior into normal and abnormal. The approach is validated by performing several controlled experiments with real trace data, obtained from programs of the Siemens test suite [59] and MySQL database server. In both scenarios, faults are injected artificially with different techniques. The approach shows very high levels of detection, with AUCROC ranging from 0.82 to 1.00.

Another supervised classification approach is proposed with Opprentice [86]. Opprentice analyzes KPI time series with Machine Learning to automatically detect anomalies online. Random Forests are applied to classify KPI point observations (composed of three variables) in normal and abnormal behavior classes, with the possibility to set up a sensitivity threshold online. The approach also includes an offline algorithm to select the optimal threshold satisfying custom linear constraints on the Precision-Recall plane. Finally, to overcome the labeling effort required by a supervised approach, a labeling tool for KPI time series like the ones observed in the paper.

Another instance of supervised anomaly detection is the TAN approach of Cohen et al. [31] (the algorithm and experimental setup are discussed in Section 4.4.2). SLO compliancy in Web servers is used as supervision data to learn a system behavioral model from combinations of metrics, applicable to perform forecasting, detection, and diagnosis. When used for online detection on synthetic workloads, the TAN approach was able to detect 91.9%–93% of failures with a false alarm rate of 6.4–16.9%.

In conclusion, we report the work of Nguyen et al. [110], later described in Section 4.4.1. While designed for black-box fault localization, the proposed system is deeply integrated with an abnormal change point detection algorithm for subsequent root-cause analysis. This detection approach

is based on Markov chains with dynamic thresholding and operates at the component level to pinpoint irregular system metric values. The Markov model is also an instance of online learning and so it is indicated for application scenarios with variable time characteristics.

4.3.2 Internet Traffic Classification (ITC). A task connected to network failure detection is **Internet Traffic Classification (ITC)**. ITC allows categorizing packets exchanged by a network system to identify network problems, to optimize resource provisioning and improve Quality-of-Service [43, 97]. It can be applied to analyze the local network flow, the incoming server requests from the outside, or the outgoing response. ITC is also widely used for cybersecurity purposes, such as intrusion detection [97, 142]. All these supported areas are connected to the appearance of failures and this fact motivates the discussion of ITC as part of failure management. ITC can be seen as a particular instance of network anomaly detection. However, in practice, the differences in methodology (emblematic of a classification problem rather than a detection problem) and data sources (network traffic rather than KPIs and metrics) justify a separate discussion.

Moore et al. [97] propose to use Supervised Machine Learning to classify the observed Internet traffic. Their work selects relevant features from a set of 248 discriminative variables (such as payload size, TCP port, etc.) to train a Naïve Bayes classifier and separate traffic according to different category sets (e.g., application-wise, maliciousness vs. legitimacy, etc.). The approach is later augmented with kernel estimation to overcome the limitations of the Gaussian assumption introduced. In a following work [8], the Bayesian framework is extended to Bayesian Neural Networks, in the architectural form of Multilayer Perceptrons, reaching 95–99% accuracy, depending on the specific test case.

Este et al. [43] develop a framework based on SVM models for TCP traffic classification. TCP communication is analyzed at the flow level rather than at the individual packet level: single-direction traffic flows between nodes are classified with multi-class kernel SVMs, depending on the application-level protocol they utilize. Single-class (one versus all) models are also trained to obtain decision boundaries and set apart outlier packets. The model surpasses correct prediction rates of 90% on three datasets under test.

Wang et al. [142] propose an end-to-end classification method for encrypted traffic based on 1D CNN. The input of the CNN is represented by raw packet data grouped according to different conventions, such as flow and session. The approach is evaluated on a public dataset (ISCX) containing both VPN and non-VPN data, where it is used to assign the observed traffic in one of the 14 categories, defined based on the application (e.g., E-mail, streaming, chat, etc.). The 1D CNN model improves the state of the art on the described dataset by approximately 10% in terms of precision and 8% in terms of recall.

4.3.3 Log Enhancement. Another task connected to failure detection is log enhancement. Its goal is to improve the quality and expressiveness of system logs, which are frequently used for detection and diagnosis tasks by IT operators and AIOps algorithms.

Zhu et al. [168] propose a logging suggestion tool, called LogAdvisor, to learn practical logging suggestions from existing log instances. In code snippets, several structural, syntactical, and textual features are extracted and filtered based on information gain, from which a decision tree is later trained to suggest logging of snippets as a binary classification problem. The approach is compared to several other baselines, including a random 50% predictor, a previous algorithmic approach called ErrLog [155], and other Machine Learning models. LogAdvisor is shown to reduce logging overhead compared to ErrLog when logging is not required, even though it is less precise and it may occasionally not place some informative print statements compared to the more conservative ErrLog.

In Reference [163], an approach for automated placement of log printing statements (Log20) based on information theory is proposed. First, entropy is shown to be an informative measure for the placement of printing statements. Then, a greedy dynamic programming algorithm for placing printing statements is implemented. The overall approach is tested on four popular distributed systems, where Log20 can be as informative as different log-level policies, while significantly reducing the print overhead (from 1.58 entries per request down to 0.08 for the INFO log level).

4.4 Root-cause Analysis (RCA)

Failure detection is the process of collection of symptoms, i.e., observations that are indicative of failures. Root-cause analysis is, instead, the process of inferring the set of faults that generated a given set of symptoms [130]. In a complex and distributed system, it is first required to isolate to restrict the analysis to the responsible component or functional subsystem, a task that we call fault localization. Only then an analysis of the possible error sources can be performed with root-cause diagnosis. As in the case of anomaly detection, we treat some associated tasks, useful to support root-cause analysis in the localization and diagnosis procedures.

4.4.1 Fault Localization. Fault localization is about identifying the set of components (devices, network links, hosts, software modules, etc.) interested by a fault that caused a specific failure. When treating this problem, it is important to clarify the scope of localization, as scale and targets may be ambiguous in this context several reasons. First, because fault localization can operate at various physical, hierarchical scales, from the entirety of a datacenter down to single chips. In large-scale services, in the presence of hundreds of thousands of machines, it is important to restrict the origin of a manifested error to the individual server level. It is also possible to perform problem determination at the physical device level, such as hard drives and processor chips. Finally, the highest level of granularity is reached by root-cause diagnosis, where the individual failure reasons are investigated (see next section). Second, from a functional perspective, fault localization may imply investigating which module in abstract, functional components (e.g., the network, or the source code) is affected by a fault. In this second perspective, the localization scale is disconnected from the physical scale. Software fault localization is applied to analyze the source code, regardless of whether the interested piece of software is deployed on thousands or tens of machines. In the network case, it may be sufficient to understand if faults are present in the network so that specialized network operators can identify the root causes independently.

Some works describe general-purpose fault localization techniques. For instance, Nguyen et al. [110] present FChain, a black-box fault localization system to pinpoint faulty components in an online cloud setting. FChain relies on low-level system metrics to detect performance anomalies, then observes how such anomaly patterns propagate to identify faulty components. Performance anomalies are identified via online metric forecasting using a discrete Markov model with an adjustable detection threshold. Then, all components displaying an anomaly are sorted by manifestation time ad examined in sequence to filter out spurious correlations according to different techniques, such as looking at interdependency between components or examining the overall propagation trend. The approach is tested on different commercial frameworks for distributed computing. In this setting, the localization performance of the approach is compared with existing black-box localization schemes, where an improvement both in terms of precision (+90%) and recall (+20%) is observed. According to the authors, FChain imposes an overhead of 1% on the cloud system performance.

Several approaches [80, 83, 132] address fault localization by correlating abnormal changes in KPI values to particular combinations of attributes (representing, e.g., geographical regions, ISPs, hosts, buckets, etc.). This combinations of values are then symptomatic of a fault for that particular

corner case. Hotspot [132] applies Monte Carlo Tree Search and hierarchical pruning to efficiently examine attribute combinations and measure how they relate to sudden changes in the Page View metric. This approach allows one to reduce hours-long manual efforts down to 20 s on average (300× speedup). Squeeze [80] proposes a similar approach based on a combined top-down, bottom-up search strategy and a newly defined correlation metric (generalized potential score, GPS). These improvements enable Squeeze to localize root causes in cases of lower statistical significance, when tested on synthetically-injected real-world datasets (F1 = 0.86–0.90). Lin et al. [83] apply pattern mining techniques on structured logs to discover association rules $X \rightarrow Y$, where Y is a predefined attribute combination describing a failure. They apply efficient processing techniques as row pre-aggregation, in-memory databases and the FP-growth mining algorithm to render the runtime complexity of association rule mining feasible. Their work also proposes five different use cases applicable in large-scale service infrastructures.

Other works apply fault localization techniques specifically to the network infrastructure of computer systems. The work of Lakhina et al. [72], already presented in Section 4.3.1 for network anomaly detection, applies the same PCA-based technique previously described to identify faulty links as well. For the diagnosis task, instead of considering the subspace distance of reconstructed data points (which was there used as a measure of anomaly), the direction of divergence is under examination. The routing matrix defines the direction of divergence from normal behavior for each link, while the most likely faulty link is selected as the one with the lowest reconstruction error after removing the estimated divergence effect caused by that link.

With Sherlock [9], Bahl et al. localize sources of performance problems in enterprise networks by constructing probabilistic inference graphs from the observation of packets exchanged in the network infrastructure. Nodes of the inference graph are divided into root-cause nodes (corresponding to internal IP entities), observation nodes (corresponding to clients), and meta-nodes, which model the dependencies between the first two types of nodes. Each node is also associated with a categorical random variable modeling the current state (up, troubled, down), which is influenced by other nodes via the dependency probabilities. Learning the inference graph corresponds to learn the dependency probabilities for each edge in the graph by observing the packets exchanged between nodes during normal operation. Once the graph is available, measurements from the observation node can be used to retrieve a set of state-node assignment vectors, corresponding to the estimated operational state of the network. According to the results of the paper, Sherlock was able to narrow down more than 87% of the failures from 350 to 16 possible root causes, while identifying 32% more faults than their previous work Shrink [64]. The same Shrink (see Section 4.4.2), although technically a root-cause diagnosis tool, can also be applied to localize physical-layer faults in networks.

Software Fault Localization (SFL) is fault localization in software components through source code analysis. An SFL system typically returns a report containing a list of suspicious statements or components. It can be considered similar to software defect prediction (Section 4.1.1), because of the similarity of investigation targets and analysis tools; however, differently from SDP, SFL relies on observed failure patterns obtained from production runs and test cases, rather than predictions of future faulty behavior based on code metrics.

Renieris et al. [121] implement a method for SFL based on similarity between program execution profiles (or *spectra*), which are obtained from execution traces to represent successful and faulty runs of programs. In the described approach, successful runs are stored as spectra in a nearest-neighbor query database. A faulty run is then used as a search term in this database, so that the nearest neighbor runs returned by the query (in particular, their differences from the faulty run in term of code coverage) can be used to produce a fault localization report with plausible target code areas responsible for the fault. The spectral representation of runs is here constituted by sets

of executed program blocks, on which the Hamming distance is computed to perform neighbor queries. In the same paper, a measure quantifying the examination effort required to find a bug, called the T-score, is introduced and it is later used by other approaches as a comparison measure [30, 85, 145].

Zeller et al. [156] propose Delta Debugging, an algorithm for the determination of state variables causing a change in the outcome of execution runs. Via graph modeling techniques, the state differences between failing and passing runs are examined to understand how state alterations influence the outcome of runs. In this way, it is possible to extract the variable interested by the fault from the variation space. In the specific case of the paper, the program state is represented as a memory graph. In a following work [30], the previously proposed search-in-space approach is complemented by a search-in-time approach, for those situations where a failure depends on when variables are assigned wrong values, an event called "cause transition." Through this consideration, programmers can reduce the amount of code that needs to be examined. During the evaluation phase conducted on the Siemens test suite [59], the **cause-transition (CT)** approach is compared to Reference [121] in terms of T-score, where CT can pinpoint defects for 5.43% of all the runs, with more than one-third (35.66%) of these runs requiring examination of less than 10% of the entire code. The described approach also provides insights about the causes behind errors and therefore supports root-cause diagnosis as well.

Liu et al. [85] adopt a statistical debugging approach (SOBER) based on the analysis of predicate evaluations in failing and passing runs. In predicate-based statistical debugging, the conditional probability of observing a failure given the observation of a particular predicate is estimated for a multitude of predicates present in the code. Predicates with a higher probability are more likely to contain a software bug or to be in the proximity of one. In this paper, a probabilistic model of predicates is introduced, including a ranking criterion to evaluate the connection of predicates to software bugs. The approach is tested on the Siemens test suite and compared using T-score and detection rate to the CT approach of Cleve and Zeller [30], pushing state-of-the-art results: when a developer is willing to analyze 1% of the interested code, SOBER can capture 8.46% of the bugs (4.65% for CT); at 20% coverage, the detection rate climbs to 73.85% (39% for CT).

Abreu et al. [1] propose a Bayesian reasoning approach (BARINEL) based on the analysis of program spectra, this time incorporated in a probabilistic framework used to estimate the health probability of components. In BARINEL, which can be applied to localize multiple faults simultaneously, coverage flags are used as a spectral representation of tests on components. A model-based reasoning system, founded on propositional logic, is constructed from the interaction between components. This model is then augmented with failure probabilities that are estimated via maximum likelihood to obtain an approximate reasoning approach, completed by a candidate ranking heuristics to deal with the high dimensionality of the problem. BARINEL can find 60% of software faults by examining less than 10% of the source code.

Wong et al. [145] propose DStar (D*), a technique to automatically suggest suspicious locations for fault localization. In particular, D* adopts a new method to compute suspiciousness coefficients from test coverage data and run outcomes. This method is based on a modification of the Kulczynski coefficients with the addition of a variable exponentiation factor (denoted by superscript *, as in the name). The approach is compared with several previous SFL techniques across 24 programs and 38 techniques, where D* is shown to be more effective on single fault localization than previous coefficient-based methods.

4.4.2 Root-cause Diagnosis. Root-cause diagnosis identifies the causes of behavior leading to failures, by recognizing the primary form of fault. For this reason, it is typically treated as a classification problem. Due to the inherent complexity and inter-dependency between components in

Table 7. Summary of Described Fault Localization Methods, Categorized by Data Sources and Target Component, with Corresponding Advantages and Disadvantages

Paper(s)	Data Sources					Targets				Advantages/Disadvantages
	Source Code	Metrics	KPIs	Logs	Netw. Traffic	Application	Hardware	Network	Datacenter	
[1, 30, 85, 121, 145, 156]	•					•				points directly to root cause/ applicable only to software bugs
[110]		•					•		•	unsupervised, online learning/ black-box monitoring
[80, 132]			•						•	general-purpose, unsupervised/ complex analysis, requires space pruning techniques
[9]			•		•		•	•		unsupervised, accurate, robust/ requires topology and traffic info
[83]				•					•	general-purpose, unsupervised/ complex analysis, requires space pruning techniques
[72]					•			•		specific/exclusive for network-traffic fault localization

software systems, it is considered a challenging task [6]. In the context of request tracing and diagnosis, the Pinpoint system [25] is one of the earliest and most salient contributions. The approach is based on the analysis of end-to-end traces of client requests, through which requests are clustered. This later enables the correlation of similar failures with components causing them. The method is tested on a simulated web e-commerce environment where artificial faults involving single and multiple components, are injected. In a following work [26] (see also Section 4.3.1), decision trees and association rules are learned to correlate failed requests with root causes. Both approaches identify the great majority of failures (93%), with variable false-positive prediction rates (23% and 50%, respectively).

Kandula et al. [64] develop Shrink, a failure diagnosis tool designed for optical links in IP networks, but certainly extendable to other networking scenarios (the authors mention, for example, the diagnosis of routers and servers). Shrink takes as input the configuration of the network (both physical and software-defined) and the current IP link status to estimate the most likely explanation for the observed failure. The diagnosis task is modeled using a bipartite Bayesian Network, connecting the set of IP links with the set of physical components on which they rely to operate; the network is then augmented with low-probability edges between unrelated nodes to improve robustness. Diagnoses are inferred by finding the maximum-likelihood state in the physical link space given the observation from the IP link status. The NP-hard search problem is approximated using a greedy inference algorithm. Since Shrink performs both detection and diagnosis, the success rate and false-positive/-negative rate are measured in the evaluation phase. Shrink excels at the diagnosis task (99.5%) in the presence of an accurate configuration description, while other approaches outperform it for detection [9].

SherLog [154] performs post-mortem analysis of logs and source code to diagnose software faults (such as code bugs and configuration errors). SherLog consists of three main building blocks: a log parser, which extract structure from log messages and identifies corresponding print statements (log points) in the source code; a path inference engine, which reconstructs execution paths

from log entries and known log points; a value inference engine, which executes the program symbolically to re-compute the value of variables along the inferred path. This type of information is directly applicable to diagnosis by programmers. SherLog is designed as a general tool, applicable without previous knowledge of the system (e.g., the structure of logs) required. Therefore its validity highly relies on the information obtainable from system logs. Moreover, the approach described is designed for single-machine, non-concurrent programs where the log information is self-enclosed in one unique source. For distributed and multi-process applications, considerations may differ.

Attariyan et al. [6] introduce X-Ray, a tool to diagnose performance anomalies in production software. X-ray implements performance summarization, a newly introduced technique for root-cause diagnosis. Performance summarization associates execution costs to fine-grained operations (such as system calls) and then assigns costs to root causes employing control flow analysis modeled ad the event level. The output corresponds to a list of root causes ranked by associated performance costs. The system is designed to identify configuration errors and user input problems. During the evaluation, measuring response time, root-cause identification ability, and necessary computation overhead, X-Ray was able to identify the first root cause in 16 of 17 tests performed on four different server frameworks (Apache, Postfix, PostgreSQL, and lighttpd), with an average runtime overhead of 2.3% in production and an average identification time of 2 min.

With their system CloudPD [127], Sharma et al. automatically construct fault signatures from the observation of past failures in the cloud. These signatures, which are composed of pairs of tracked variables and thresholds, are used at runtime to diagnose problems identified with the behavioral modeling engine (see Section 4.3.1) previously discussed. The system can diagnose several cloud and VM-related anomalies, such as invalid resource sizing, VM faulty reconfiguration, or workload mix change. The accuracy of their approach ranges between 83% and 88%, depending on the employed benchmark.

Samir et al. [124] utilize **Hierarchical Hidden Markov Models (HHMM)** to associate resource anomalies to root causes in clustered resource environments, on the different levels of container, node, and cluster. Markov models are constructed on different levels, trained with the Baum-Welch algorithm using response time sequences as observations. The identification approach is evaluated in terms of accuracy and compared to two other algorithms. The predictions obtained during the identification step are also fed into a recovery component for automatic healing (see Section 4.5.3).

In conclusion, the already mentioned approach of Cohen et al. [31] (see Sections 4.2.2 and 4.3.1) can also be used to diagnose observed failures. The described TAN model correlates observable metrics to SLO violations. If during the diagnosis the same input metrics are measured, then it is possible to use the probabilistic model to obtain a list of metrics correlating with specific events or problems. While this information may not always constitute the final diagnosis, the approach supports human diagnosis with viable hypotheses and evidence.

4.4.3 Other Tools Supporting RCA. This section analyses other smart software tools that, although they do not diagnose or localize faulty behavior, can assist operators and developers while investigating detected problems and can, therefore be seen as ancillary resources for the main root-cause analysis task [3]. These tools may include, for example, information retrieval mechanisms to quickly gather evidence of recurrent problems, like their relative frequency; or dependency models for distributed systems used to understand causal relationships between events and/or components to accelerate future diagnoses.

Aguilera et al. [3] describe several approaches to identify performance problems in distributed systems by analyzing causal path dependencies between black-box components. Causal path patterns are extracted from message traces using two different algorithms, the first applying a search

heuristic on single messages to identify nested call pairs, the second based on signal processing. The algorithms are compared in different applicability scenarios and tested on different categories of traces, both synthetic and real. The off-line analysis proposed, while not directly focusing on failures, can prove useful from the perspective of IT operators, willing to acquire a component dependency model. Such a model can therefore represent a valuable tool to diagnose failures. Similar conclusions can be drawn for other dependency modeling approaches [9, 29].

Podgurski et al. [120] propose to use Machine Learning algorithms to classify and group software failures to facilitate error prioritization and diagnosis. Programs are instrumented to collect execution profile data, which are then used to train a logistic regression classifier to predict failure causes. The feature coefficients obtained during this training procedure are used to select relevant features for the clustering task, where related failures are grouped using the k-medoids algorithm. In the experimental phase, failures collected from three large compiler programs are grouped according to the described method and the corresponding clusters are visualized in combination with results from the classification phase. The experimental evaluation shows, with various experiments, how for a large number of clusters ($\geq$71%) the majority of failures originated from the same cause were assigned to the same cluster.

Cohen et al. [32] present a method for extracting signatures describing the state of a running system, which can be later used to retrieve similar previous states. The same method can be used to group similar problems and quantify the frequency of recurring problems for prioritization purposes. Starting from the observation that recording raw system metrics as state representation is ineffective from a retrieval perspective, in the proposed approach state signatures incorporate information about the relation between the **Service-level Agreement (SLA)** state and the abnormality level of the metrics measured during the state observation. In particular, relevant metrics are selected and their values are characterized as typical or atypical, by learning a joint descriptive model of metric and SLA state (as previously done in Reference [31]). Individual metrics are also marked as attributed, not attributed, or irrelevant to the SLA state, based on the interpretation given by the model for each specific metric. These flags constitute the final signature representation, in combination with the SLA state predicted by the model. The signature representations of states are clustered using the k-means and k-medians algorithms. The approach is validated on traces from two distributed applications, serving synthetic and real production workloads, showing the retrieval system to be successful in recognizing different performance problems. A similar approach is proposed in Reference [14], where the values of performance metrics are categorized in different classes ("hot," "cold," and "normal") based on their quantile distribution. Relevant metrics are selected by measuring their predictive power when training a logistic regression classifier to predict SLA violations. Fingerprints are constructed both at the epoch and crisis level, indicating different durations in time for non-conformant episodes, and are compared using distance functions. The approach can associate new crises to past behavior in 80% of the cases with an average running time of 10 min.

With LogCluster [84], Lin et al. use a knowledge base, obtained via clustering, to quickly retrieve logs of recurrent problems and facilitate problem identification. Logs, which are interpreted as sequences of discrete log events, are first transformed to a vector representation using **Inverse Document Frequency (IDF)**, where events represent terms. Then, logs are clustered using agglomerative hierarchical clustering and one element for each cluster is selected as the representative instance. At runtime, the system performs queries on the restricted set of representatives, rather than the entire knowledge base, reducing efforts for similarity-based retrieval. In the evaluation performed on Hadoop applications (WordCount, PageRank) and Microsoft internal services, LogCluster outperforms previous approaches in retrieval relevance (measured in precision) and number of log sequences needed to be examined to identify failures.

4.5 Remediation

Thanks to the problem-specific knowledge gathered during the diagnosis step, like the identification of root causes or the isolation of a faulty component, it is possible to initiate a sequence of automatic repair actions, which are here described as remediation. Remediation approaches are often linked with certain concepts of service desk management, such as ticket routing or ticket solution recommendation. Remediation has experienced less scientific contributions linked with AI compared to the prevention, prediction, detection, and diagnosis tasks. This is possibly due to the above-mentioned fact that, once the nature of the underlying problem has been clarified through diagnosis, the recovery steps are almost immediately identifiable and attainable without resorting to complex models. We divide the available contributions into three categories, constituting successive steps toward resolution: incident triage, solution recommendation, and recovery.

4.5.1 Incident Triage. Incident triage is the step in problem resolution dealing with categorizing a reported problem. The purpose of triage is often the assignment to the correct expert resolution group [126, 158]. Triage can also be used to select a suitable diagnosis and remediation algorithm.

Shao et al. [126] propose to mine resolution sequences, i.e., the steps followed by a problem ticket from reporting to resolution, to improve future ticket routing and speed up recovery. Ticket routing sequences are analyzed employing a Markov Model, from which several routing algorithms are developed and later tested in terms of effectiveness, robustness, and applicability. The Mean number of Steps to Resolve is reduced from 3.94 to 2.58 (−34.52%) on 2,634 tickets. The approach is entirely based on the observation of resolution sequences and does not rely on ticket content.

Zeng et al. [158], however, propose with Kilo to classify ticket data on multiple levels by analyzing symptom descriptions of problems. Their work deals with the ticket labeling problem from a hierarchical perspective, where tickets are assigned to increasingly specific subclasses in a tree hierarchy. The classification algorithm, based on Bayesian decision theory, introduces a new hierarchical loss function minimizing the expected misclassification risk when making non-leaf predictions in the tree label structure. A greedy prediction algorithm is deployed to perform hierarchical classification, with the ability to integrate available domain expert knowledge in the form of prior probabilities.

4.5.2 Solution Recommendation. The approaches here described provide implementations for recommending solutions to occurring problems. Most solutions are based on past incident history and rely on the annotation of solutions in a previous resolution window. Therefore, these approaches mostly operate as retrieval systems that resemble the ones described for root-cause analysis in Section 4.4.3. In the presence of annotated solutions, those approaches can equally be a viable solution for this task.

Zhou et al. [166] propose similarity-based algorithms to suggest the resolution of repeating problems from incident tickets. The basic approach consists in retrieving k ticket resolution suggestions using a k-NN approach. The similarity between tickets is evaluated based on a mixture of the available numerical, categorical, and textual data, for which individual and aggregate similarity measures are defined. The basic solution is extended to deal with the problem of false-positive tickets, present in the historical ticket data as well in the incoming tickets requiring a new solution. To solve this, each ticket is classified as real or false using a binary classifier with a k-NN approach and then by weighing ticket importance based on the prediction outcome. The final solution recommendation takes into account both importance and similarity. The paper also incorporates other ideas for improving feature extraction such as topic discovery and metric learning.

Wang et al. [140] propose a cognitive framework based on the use of ontologies to construct domain-specific knowledge and suggest recovery actions for tickets in IT service management.

The approach is based on the analysis of free-form text present in summary and resolution descriptions in tickets. First, domain-specific phrases are extracted from text fields using different language processing techniques; then, an ontology model is developed to provide definitions, classes, and interconnecting relations of keywords, for which a hierarchy is also established; the obtained model can then be used to recommend resolution actions by matching concept patterns extracted in incoming and historical tickets via similarity functions (such as the Jaccard distance). In the experimental phase, the extracting accuracy of concept patterns is tested with ground truth labels, reaching the value of 86.2% for the prediction of required actions.

In a work presented by Facebook [82], natural language processing techniques are deployed to predict repair actions from closed tickets. Up to five repair actions are recommended by analyzing raw text logs as input features, with accuracy ranging from 50% to 80% (no experimental evaluation is described). In the same paper, other in-house failure management systems are illustrated, including an online anomaly detection algorithm and an automatic repair engine.

4.5.3 Recovery. We define as recovery approaches those methods taking direct and independent actions toward the resolution of a diagnosed problem. According to the analysis derived from our mapping study, no distinctive contribution has been proposed to perform direct recovery actions with AI. The closest match is represented by the work of Samir et al. [124], where a combined detection/RCA/remediation framework associates detected anomalies to root causes utilizing HHMM (see Section 4.4.2). The last step of their pipeline consists in performing recovery actions based on the insights obtained in the identification step. Specific recovery actions are defined for each failure case in a predefined manner. The efficiency of the recovery step is assessed via MTTR and recovery rate.

By applying the term in a broader sense, we may consider as recovery all those preventive actions that do not require diagnosis information to be undertaken. This is the case, for example of reactive mechanisms applied in combination with online failure prediction mechanisms (described in Section 4.2), to perform data migration or checkpointing [33, 78] preemptively.

5 CONCLUSION

5.1 Current and Future Trends in Failure Management

In the previous section, we described many AI contributions to deal with failures in AIOps. In this final section, we focus on the large picture and analyze the current status of failure management of AIOps. We also make use of our observations to examine the currently open challenges suggest future directions for research.

AIOps has shown a steady growing trend in the last years, manifested both in the number and variety of contributions present. In the last five years, at least 100 contributions have been proposed on a yearly average. We can expect the field to continue its growth, due to increasing demand for reliability and efficiency in large-scale computing systems. The evolution of cloud technologies (e.g., in virtualization, monitoring tools) will provide large space for future improvements and the experimentation of new techniques. To fulfill these great expectations, the field must be able to provide a solid ground for experimentation, based on a more formal standardization of problems and stronger attitude toward benchmarks, needed for comparison and evaluation of the results achieved. Efforts in creating standard problems and benchmark datasets would therefore be rewarding.

The analysis of the topics and tasks in the current AIOps landscape, as observed and described in Section 4, equally allows one to investigate possible future directions. First, Table 8 shows how the majority of works utilize only one or few types of data. Multimodal approaches, able to take

Table 8. Papers Analyzed in the Survey Grouped by Employed Data Sources, Targets, and Categories

Paper(s)	Data Sources									Targets					Category
	Source Code	Testing Resources	System Metrics	KPIs/SLO data	Network Traffic	Topology	Incident Reports	Event Logs	Execution Traces	Source Code	Application	Hardware	Network	Datacenter	
[35, 38, 52, 94, 102, 117, 153]	•									•					Software Defect Prediction
[42, 50, 76, 98, 103, 115, 141]	•									•					Software Defect Prediction
[105, 128]	•	•								•					Fault Injection
[4, 49, 136]			•								•			•	Software Rejuvenation
[21, 137]			•	•							•			•	Software Rejuvenation
[62, 95, 114]											•			•	Checkpointing
[77, 78, 89, 104, 138]			•									•			Hardware Failure Prediction
[54, 88, 143, 165, 167]			•									•			Hardware Failure Prediction
[101, 148, 149, 164]			•									•			Hardware Failure Prediction
[37]			•									•		•	Hardware Failure Prediction
[33]			•					•				•			Hardware Failure Prediction
[161]								•				•	•		Hardware Failure Prediction
[23]			•								•			•	System Failure Prediction
[31]			•	•							•				System Failure Prediction
[61]			•	•							•			•	System Failure Prediction
[119]			•			•					•			•	System Failure Prediction
[81]								•			•			•	System Failure Prediction
[123]								•			•	•	•	•	System Failure Prediction
[44, 160]								•			•				System Failure Prediction
[151]	•							•						•	Anomaly Detection
[7, 131, 159]			•									•		•	Anomaly Detection
[86, 150]				•							•				Anomaly Detection
[127]			•	•							•			•	Anomaly Detection
[72]					•	•							•		Anomaly Detection
[73]					•	•					•		•		Anomaly Detection
[12, 18, 41, 45, 92, 162]								•			•				Anomaly Detection
[29]								•	•		•			•	Anomaly Detection
[11]									•		•			•	Anomaly Detection
[26, 87]									•		•				Anomaly Detection
[8, 43, 97, 142]					•								•	•	Internet Traffic Classification
[163, 168]	•							•						•	Log Enhancement
[1, 30, 85, 121, 145, 156]	•	•								•	•				Fault Localization
[110]			•			•						•		•	Fault Localization
[80, 132]				•										•	Fault Localization
[83]								•						•	Fault Localization
[9]				•	•							•	•		Fault Localization
[154]	•							•			•				Root-cause Diagnosis
[6]	•		•	•							•				Root-cause Diagnosis
[64]					•	•							•		Root-cause Diagnosis
[25]									•		•			•	Root-cause Diagnosis
[120]		•				•					•				RCA - Others
[14, 32]			•	•							•			•	RCA - Others
[84]								•			•			•	RCA - Others
[3]									•		•			•	RCA - Others
[126, 158]							•				•			•	Incident Triage
[140, 166]							•				•			•	Solution Recommendation
[82]							•	•			•			•	Solution Recommendation
[124]			•	•							•			•	Recovery

advantage of different data sources, may prove more effective and robust to new observations, thanks to the increase in system visibility.

We also observed how some areas of failure management have experienced less scientific interest compared to others. A clear example is the recovery task, which, although a fundamental and concrete step to deal with failures, still presents a minute group of contributions. A similar consideration applies for failure prevention, where all the contributions are concentrated around few tasks (we presented four subcategories). However, failure prevention can be performed in many other ways, some of which are still to be explored. Currently, the majority of approaches for failure prevention are applied online and concentrate exclusively on the current-future state characteristics. Introducing assumptions and information about the system working principles may set the ground for much more actionable insights. For example, model-based prevention would allow operators to estimate in advance the risks associated with particular actions, such as a canary release or a server shutdown.

In addition, the advent of virtualization technologies requires new research focusing on specific targets (e.g., hypervisors, virtual machines, containers, etc.), creating new tasks, such as hypervisor anomaly detection, container failure prediction and so on.

Finally, the application of novel AI approaches may prove beneficial to advance AIOps. In the decade, the rise of Deep Learning methods has translated into a variety of new approaches for failure prediction, anomaly detection and root-cause analysis. This may occur again with future breakthroughs.

5.2 Concluding Remarks and Future Work

In this work, we explored a variety of AIOps approaches for the management of failures in IT systems. To conduct our research, the AIOps concept has been explored starting from the available definitions, allowing us to capture a precise characterization of the topic in terms of goals, sources, and methods. Our survey study focusing on failure management analyzed 100 contributions, across all identified categories, data sources, and target components. When possible, we have focused our attention on the limitations of current approaches and the possibilities to expand and integrate the current areas of research in AIOps. We hope that the results here presented can actively support researchers and engineers working with AIOps by providing a comprehensive overview of the topic to support future investigations. As AIOps is an active and increasingly popular research area, we expect future works to update this article with new references and contribute to expanding the discussion with newly established topics.

REFERENCES

[1] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. 2009. Spectrum-based multiple fault localization. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 88–99. https://doi.org/10.1109/ASE.2009.25

[2] Armen Aghasaryan, Eric Fabre, Albert Benveniste, Renée Boubour, and Claude Jard. 1998. Fault detection and diagnosis in distributed systems: An approach by partially stochastic petri nets. *Discrete Event Dynam. Syst.* 8, 2 (1998), 203–231. https://doi.org/10.1023/a:1008241818642

[3] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. 2003. Performance debugging for distributed systems of black boxes. *ACM SIGOPS Operat. Syst. Rev.* 37, 5 (Dec. 2003), 74–89. https://doi.org/10.1145/1165389.945454

[4] Javier Alonso, Jordi Torres, Josep Ll. Berral, and Ricard Gavalda. 2010. Adaptive on-line software aging prediction based on machine learning. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems Networks (DSN'10)*. IEEE, 507–516. https://doi.org/10.1109/dsn.2010.5544275

[5] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, and D. Powell. 1990. Fault injection for dependability validation: A methodology and some applications. *IEEE Trans. Softw. Eng.* 16, 2 (Feb. 1990), 166–182. https://doi.org/10.1109/32.44380

[6] Mona Attariyan, Michael Chow, and Jason Flinn. 2012. X-Ray: Automating root-cause diagnosis of performance anomalies in production software. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*. USENIX Association, 307–320. https://doi.org/10.5555/2387880.2387910

[7] Julien Audibert, Pietro Michiardi, Frédéric Guyard, Sébastien Marti, and Maria A. Zuluaga. 2020. USAD: UnSupervised anomaly detection on multivariate time series. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'20)*. ACM, New York, NY, 3395–3404. https://doi.org/10.1145/3394486.3403392

[8] Tom Auld, Andrew W. Moore, and Stephen F. Gull. 2007. Bayesian neural networks for internet traffic classification. *IEEE Trans. Neural Netw.* 18, 1 (Jan. 2007), 223–239. https://doi.org/10.1109/tnn.2006.883010

[9] Paramvir Bahl, Ranveer Chandra, Albert Greenberg, Srikanth Kandula, David A. Maltz, and Ming Zhang. 2007. Towards highly reliable enterprise network services via inference of multi-level dependencies. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM'07)*. ACM, New York, NY, 13–24. https://doi.org/10.1145/1282380.1282383

[10] Chetan Bansal, Sundararajan Renganathan, Ashima Asudani, Olivier Midy, and Mathru Janakiraman. 2020. DeCaf: Diagnosing and triaging performance issues in large-scale cloud services. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP'20)*. ACM, New York, NY, 201–210. https://doi.org/10.1145/3377813.3381353

[11] Paul Barham, Rebecca Isaacs, Richard Mortier, and Dushyanth Narayanan. 2003. Magpie: Online modelling and performance-aware systems. In *Proceedings of the 9th Conference on Hot Topics in Operating Systems*, Vol. 9. USENIX Association, 15. https://doi.org/10.5555/1251054.1251069

[12] Ivan Beschastnikh, Yuriy Brun, Michael D. Ernst, and Arvind Krishnamurthy. 2014. Inferring models of concurrent systems from logs of their behavior with CSight. In *Proceedings of the 36th International Conference on Software Engineering (ICSE'14)*. ACM, New York, NY, 468–479. https://doi.org/10.1145/2568225.2568246

[13] Netflix Technology Blog. 2016. Netflix Chaos Monkey Upgraded. Retrieved from https://netflixtechblog.com/netflix-chaos-monkey-upgraded-1d679429be5d.

[14] Peter Bodik, Moises Goldszmidt, Armando Fox, Dawn B. Woodard, and Hans Andersen. 2010. Fingerprinting the datacenter: Automated classification of performance crises. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys'10)*. ACM, New York, NY, 111–124. https://doi.org/10.1145/1755913.1755926

[15] A. T. Bouloutas, S. Calo, and A. Finkel. 1994. Alarm correlation and fault identification in communication networks. *IEEE Trans. Commun.* 42, 2/3/4 (2 1994), 523–533. https://doi.org/10.1109/tcomm.1994.577079

[16] L. C. Briand, J. W. Daly, and J. K. Wust. 1999. A unified framework for coupling measurement in object-oriented systems. *IEEE Trans. Softw. Eng.* 25, 1 (1 1999), 91–121. https://doi.org/10.1109/32.748920

[17] Broadcom. 2020. AIOps—Broadcom. Retrieved from https://www.broadcom.com/products/software/aiops.

[18] Andy Brown, Aaron Tuor, Brian Hutchinson, and Nicole Nichols. 2018. Recurrent neural network attention mechanisms for interpretable system log anomaly detection. In *Proceedings of the 1st Workshop on Machine Learning for Computing Systems (MLCS'18)*. ACM, New York, NY, Article 1, 8 pages. https://doi.org/10.1145/3217871.3217872

[19] Lisa Burnell and Eric Horvitz. 1995. Structure and chance: Melding logic and probability for software debugging. *Commun. ACM* 38, 3 (Mar. 1995), 31–ff. https://doi.org/10.1145/203330.203338

[20] K. L. Butler and J. A. Momoh. 1999. A neural net based approach for fault diagnosis in distribution networks. In *Proceedings of the IEEE Power Engineering Society*, Vol. 1. IEEE, 353–356. https://doi.org/10.1109/PESW.1999.747478

[21] V. Castelli, R. E. Harper, P. Heidelberger, S. W. Hunter, K. S. Trivedi, K. Vaidyanathan, and W. P. Zeggert. 2001. Proactive management of software aging. *IBM J. Res. Dev.* 45, 2 (Mar. 2001), 311–332. https://doi.org/10.1147/rd.452.0311

[22] Raghavendra Chalapathy and Sanjay Chawla. 2019. Deep Learning for Anomaly Detection: A Survey. Retrieved from http://arxiv.org/abs/1901.03407.

[23] Thanyalak Chalermarrewong, Tiranee Achalakul, and Simon Chong Wee See. 2012. Failure prediction of data centers using time series and fault tree analysis. In *Proceedings of the IEEE 18th International Conference on Parallel and Distributed Systems*. IEEE, 794–799. https://doi.org/10.1109/icpads.2012.129

[24] Varun Chandola, Arindam Banerjee, and Vipin Kumar. 2009. Anomaly detection: A survey. *Comput. Surveys* 41, 3 (July 2009), 15:1–15:58. https://doi.org/10.1145/1541880.1541882

[25] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. 2002. Pinpoint: Problem determination in large, dynamic Internet services. In *Proceedings of the International Conference on Dependable Systems and Networks*. IEEE, 595–604. https://doi.org/10.1109/DSN.2002.1029005

[26] Mike Y. Chen, Anthony Accardi, Emre Kiciman, Jim Lloyd, Dave Patterson, Armando Fox, and Eric Brewer. 2004. Path-based failure and evolution management. In *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation (NSDI'04)*. USENIX Association, 23. Retrieved from https://dl.acm.org/doi/10.5555/1251175.1251198.

[27] Xin Chen, Charng-Da Lu, and Karthik Pattabiraman. 2014. Failure analysis of jobs in compute clouds: A Google cluster case study. In *Proceedings of the IEEE 25th International Symposium on Software Reliability Engineering*. IEEE, 167–177. https://doi.org/10.1109/issre.2014.34

[28] S. R. Chidamber and C. F. Kemerer. 1994. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.* 20, 6 (6 1994), 476–493. https://doi.org/10.1109/32.295895

[29] Michael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas F. Wenisch. 2014. The mystery machine: End-to-end performance analysis of large-scale internet services. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. USENIX Association, 217–231. https://dl.acm.org/doi/10.5555/2685048.2685066

[30] Holger Cleve and Andreas Zeller. 2005. Locating causes of program failures. In *Proceedings of the 27th International Conference on Software Engineering (ICSE'05)*. ACM, New York, NY, 342–351. https://doi.org/10.1145/1062455.1062522

[31] Ira Cohen, Moises Goldszmidt, Terence Kelly, Julie Symons, and Jeffrey S. Chase. 2004. Correlating instrumentation data to system states: A building block for automated diagnosis and control. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design and Implementation (OSDI'04)*. USENIX Association, 16. Retrieved from https://dl.acm.org/doi/10.5555/1251254.1251270.

[32] Ira Cohen, Steve Zhang, Moises Goldszmidt, Julie Symons, Terence Kelly, and Armando Fox. 2005. Capturing, indexing, clustering, and retrieving system history. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP'05)*. ACM, New York, NY, 105–118. https://doi.org/10.1145/1095810.1095821

[33] Carlos H. A. Costa, Yoonho Park, Bryan S. Rosenburg, Chen-Yong Cher, and Kyung Dong Ryu. 2014. A system software approach to proactive memory-error avoidance. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'14)*. IEEE, 12 pages. https://doi.org/10.1109/SC.2014.63

[34] A. Csenki. 1990. Bayes predictive analysis of a fundamental software reliability model. *IEEE Trans. Reliabil.* 39, 2 (June 1990), 177–183. https://doi.org/10.1109/24.55879

[35] Marco D'Ambros, Michele Lanza, and Romain Robbes. 2011. Evaluating defect prediction approaches: A benchmark and an extensive comparison. *Empir. Softw. Eng.* 17, 4–5 (Aug. 2011), 531–577. https://doi.org/10.1007/s10664-011-9173-9

[36] Yingnong Dang, Qingwei Lin, and Peng Huang. 2019. AIOps: Real-world challenges and research innovations. In *Proceedings of the IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE'19)*. IEEE, 4–5. https://doi.org/10.1109/icse-companion.2019.00023

[37] Nickolas Allen Davis, Abdelmounaam Rezgui, Hamdy Soliman, Skyler Manzanares, and Milagre Coates. 2017. FailureSim: A system for predicting hardware failures in cloud data centers using neural networks. In *Proceedings of the IEEE 10th International Conference on Cloud Computing (CLOUD'17)*. IEEE, 544–551. https://doi.org/10.1109/cloud.2017.75

[38] Karel Dejaeger, Thomas Verbraken, and Bart Baesens. 2013. Toward comprehensible software fault prediction models using Bayesian network classifiers. *IEEE Trans. Softw. Eng.* 39, 2 (Feb. 2013), 237–257. https://doi.org/10.1109/tse.2012.20

[39] B. Dhanalaxmi, G. Apparao Naidu, and K. Anuradha. 2015. A review on software fault detection and prevention mechanism in software development activities. *J. Comput. Eng.* 17, 6 (2015), 25–30.

[40] Peter A. Dinda and David R. O'Hallaron. 1999. An evaluation of linear models for host load prediction. In *Proceedings of The 8th International Symposium on High Performance Distributed Computing*. IEEE, 87–96.

[41] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. 2017. DeepLog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1285–1298. https://doi.org/10.1145/3133956.3134015

[42] Karim O. Elish and Mahmoud O. Elish. 2008. Predicting defect-prone software modules using support vector machines. *J. Syst. Softw.* 81, 5 (May 2008), 649–660. https://doi.org/10.1016/j.jss.2007.07.040

[43] Alice Este, Francesco Gringoli, and Luca Salgarelli. 2009. Support vector machines for TCP traffic classification. *Comput. Netw.* 53, 14 (Sep. 2009), 2476–2490. https://doi.org/10.1016/j.comnet.2009.05.003

[44] Ilenia Fronza, Alberto Sillitti, Giancarlo Succi, Mikko Terho, and Jelena Vlasenko. 2013. Failure prediction based on log files using random indexing and support vector machines. *J. Syst. Softw.* 86, 1 (1 2013), 2–11. https://doi.org/10.1016/j.jss.2012.06.025

[45] Qiang Fu, Jian-Guang Lou, Yi Wang, and Jiang Li. 2009. Execution anomaly detection in distributed systems through unstructured log analysis. In *Proceedings of the 9th IEEE International Conference on Data Mining*. IEEE Computer Society, 149–158. https://doi.org/10.1109/icdm.2009.60

[46] Zhiwei Gao, Carlo Cecati, and Steven X. Ding. 2015. A survey of fault diagnosis and fault-tolerant techniques—Part I: Fault diagnosis with model-based and signal-based approaches. *IEEE Trans. Industr. Electr.* 62, 6 (June 2015), 3757–3767. https://doi.org/10.1109/tie.2015.2417501

[47] Zhiwei Gao, Carlo Cecati, and Steven X. Ding. 2015. A survey of fault diagnosis and fault-tolerant techniques—Part II: Fault diagnosis with knowledge-based and hybrid/active approaches. *IEEE Trans. Industr. Electr.* 62, 6 (June 2015), 3768–3774. https://doi.org/10.1109/TIE.2015.2419013

[48] S. Garg, A. Puliafito, M. Telek, and K. S. Trivedi. 1995. Analysis of software rejuvenation using Markov regenerative stochastic petri net. In *Proceedings of the 6th International Symposium on Software Reliability Engineering (ISSRE'95)* (1995). IEEE, 180–187. https://doi.org/10.1109/issre.1995.497656

[49] S. Garg, A. van Moorsel, K. Vaidyanathan, and K. S. Trivedi. 1998. A methodology for detection and estimation of software aging. In *Proceedings of the 9th International Symposium on Software Reliability Engineering*. IEEE, 283–292. https://doi.org/10.1109/issre.1998.730892

[50] Emanuel Giger, Marco D'Ambros, Martin Pinzger, and Harald C. Gall. 2012. Method-level bug prediction. In *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM'12)*. ACM, New York, NY, 171–180. https://doi.org/10.1145/2372251.2372285

[51] Tarun Goyal, Ajit Singh, and Aakanksha Agrawal. 2012. Cloudsim: Simulator for cloud computing infrastructure and modeling. *Proc. Eng.* 38 (Nov. 2012), 3566–3572. https://doi.org/10.1016/j.proeng.2012.06.412

[52] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy. 2000. Predicting fault incidence using software change history. *IEEE Trans. Softw. Eng.* 26, 7 (July 2000), 653–661. https://doi.org/10.1109/32.859533

[53] Maurice H. Halstead. 1977. *Elements of Software Science (Operating and Programming Systems Series)*. Elsevier Science Inc..

[54] Greg Hamerly and Charles Elkan. 2001. Bayesian approaches to failure prediction for disk drives. In *Proceedings of the 18th International Conference on Machine Learning (ICML'01)*. Morgan Kaufmann, San Francisco, CA, 202–209. https://doi.org/10.5555/645530.655825

[55] Seungjae Han, K. G. Shin, and H. A. Rosenberg. 1995. DOCTOR: An integrated software fault injection environment for distributed real-time systems. In *Proceedings of the IEEE International Computer Performance and Dependability Symposium*. IEEE, 204–213. https://doi.org/10.1109/IPDS.1995.395831

[56] J. L. Hellerstein, Fan Zhang, and P. Shahabuddin. 1999. An approach to predictive detection for service management. In *Proceedings of the 6th IFIP/IEEE International Symposium on Integrated Network Management*. IEEE, 309–322. https://doi.org/10.1109/inm.1999.770691

[57] Y. Huang, C. Kintala, N. Kolettis, and N. D. Fulton. 1995. Software rejuvenation: Analysis, module and applications. In *Proceedings of the 25th International Symposium on Fault-Tolerant Computing. Digest of Papers*. IEEE, 381–390. https://doi.org/10.1109/FTCS.1995.466961

[58] G. F. Hughes, J. F. Murray, K. Kreutz-Delgado, and C. Elkan. 2002. Improved disk-drive failure warnings. *IEEE Trans. Reliabil.* 51, 3 (Sep. 2002), 350–357. https://doi.org/10.1109/TR.2002.802886

[59] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. 1994. Experiments on the effectiveness of dataflow- and control-flow-based test adequacy criteria—IEEE conference publication. In *Proceedings of the 16th International Conference on Software Engineering*. IEEE, 191–200. Retrieved from https://ieeexplore.ieee.org/document/296778.

[60] Olumuyiwa Ibidunmoye, Francisco Hernández-Rodriguez, and Erik Elmroth. 2015. Performance anomaly detection and bottleneck identification. *Comput. Surveys* 48, 1 (Sep. 2015), 1–35. https://doi.org/10.1145/2791120

[61] Tariqul Islam and Dakshnamoorthy Manivannan. 2017. Predicting application failure in cloud: A machine learning approach. In *Proceedings of the IEEE International Conference on Cognitive Computing (ICCC'17)*. IEEE, 24–31. https://doi.org/10.1109/ieee.iccc.2017.11

[62] Itthichok Jangjaimon and Nian-Feng Tzeng. 2015. Effective cost reduction for elastic clouds under spot instance pricing through adaptive checkpointing. *IEEE Trans. Comput.* 64, 2 (Feb. 2015), 396–409. https://doi.org/10.1109/tc.2013.225

[63] David Jauk, Dai Yang, and Martin Schulz. 2019. Predicting faults in high performance computing systems: An in-depth survey of the state-of-the-practice. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'19)*. ACM, New York, NY, Article 30, 13 pages. https://doi.org/10.1145/3295500.3356185

[64] Srikanth Kandula, Dina Katabi, and Jean-Philippe Vasseur. 2005. Shrink: A tool for failure diagnosis in IP networks. In *Proceeding of the ACM SIGCOMM Workshop on Mining Network Data (MineNet'05)*. ACM, New York, NY, 6 pages. https://doi.org/10.1145/1080173.1080178

[65] N. Karunanithi, D. Whitley, and Y. K. Malaiya. 1992. Prediction of software reliability using connectionist models. *IEEE Trans. Softw. Eng.* 18, 7 (July 1992), 563–574. https://doi.org/10.1109/32.148475

[66] Taghi M. Khoshgoftaar and David L. Lanning. 1995. A neural network approach for early detection of program modules having high risk in the maintenance phase. *J. Syst. Softw.* 29, 1 (Apr. 1995), 85–91. https://doi.org/10.1016/0164-1212(94)00130-f

[67] Barbara A. Kitchenham, David Budgen, and O. Pearl Brereton. 2010. The value of mapping studies - A participant-observer case study. In *Proceedings of the 14th International Conference on Evaluation and Assessment in Software Engineering (EASE'10)*. BCS Learning & Development, 25–33. https://doi.org/10.14236/ewic/EASE2010.4

[68] S. Kliger, S. Yemini, Y. Yemini, D. Ohsie, and S. Stolfo. 1995. *A Coding Approach to Event Correlation*. Springer US, Boston, MA, 266–277. https://doi.org/10.1007/978-0-387-34890-2_24

[69] Yevgeniy Sverdlik Data Center Knowledge. 2016. What Facebook Has Learned from Regularly Shutting Down Entire Data Centers. Retrieved from https://www.datacenterknowledge.com/archives/2016/08/31/facebook-learned-regularly-shutting-entire-data-centers.

[70] Khairy A. H. Kobbacy and Sunil Vadera. 2011. A survey of AI in operations management from 2005 to 2009. *J. Manufact. Technol. Manage.* 22, 6 (July 2011), 706–733. https://doi.org/10.1108/17410381111149602

[71] K. A. H. Kobbacy, S Vadera, and M. H. Rasmy. 2007. AI and OR in management of operations: History and trends. *J. Oper. Res. Soc.* 58, 1 (Jan. 2007), 10–28. https://doi.org/10.1057/palgrave.jors.2602132

[72] Anukool Lakhina, Mark Crovella, and Christophe Diot. 2004. Diagnosing network-wide traffic anomalies. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM'04)*. ACM, New York, NY, 219–230. https://doi.org/10.1145/1015467.1015492

[73] Anukool Lakhina, Mark Crovella, and Christophe Diot. 2005. Mining anomalies using traffic feature distributions. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (Philadelphia, Pennsylvania) *(SIGCOMM'05)*. ACM, New York, NY, 217–228. https://doi.org/10.1145/1080091.1080118

[74] Andrew Lerner. 2017. AIOps Platforms—Gartner. Retrieved from https://blogs.gartner.com/andrew-lerner/2017/08/09/aiops-platforms/.

[75] Anna Levin, Shelly Garion, Elliot K. Kolodner, Dean H. Lorenz, Katherine Barabash, Mike Kugler, and Niall McShane. 2019. AIOps for a cloud object storage service. In *Proceedings of the IEEE International Congress on Big Data (BigDataCongress'19)*. IEEE, 165–169. https://doi.org/10.1109/BigDataCongress.2019.00036

[76] Jian Li, Pinjia He, Jieming Zhu, and Michael R. Lyu. 2017. Software defect prediction via convolutional neural network. In *Proceedings of the IEEE International Conference on Software Quality, Reliability and Security (QRS'17)*. IEEE, 318–328. https://doi.org/10.1109/qrs.2017.42

[77] Jing Li, Xinpu Ji, Yuhan Jia, Bingpeng Zhu, Gang Wang, Zhongwei Li, and Xiaoguang Liu. 2014. Hard drive failure prediction using classification and regression trees. In *Proceedings of the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 383–394. https://doi.org/10.1109/dsn.2014.44

[78] Jing Li, Rebecca J. Stones, Gang Wang, Zhongwei Li, Xiaoguang Liu, and Kang Xiao. 2016. Being accurate is not enough: New metrics for disk failure prediction. In *Proceedings of the IEEE 35th Symposium on Reliable Distributed Systems (SRDS'16)*. IEEE, 71–80. https://doi.org/10.1109/srds.2016.019

[79] Yangguang Li, Zhen Ming (Jack) Jiang, Heng Li, Ahmed E. Hassan, Cheng He, Ruirui Huang, Zhengda Zeng, Mian Wang, and Pinan Chen. 2020. Predicting node failures in an ultra-large-scale cloud computing platform: An AIOps solution. *ACM Trans. Softw. Eng. Methodol.* 29, 2 (27 4 2020), 13:1–13:24. https://doi.org/10.1145/3385187

[80] Zeyan Li, Chengyang Luo, Yiwei Zhao, Yongqian Sun, Kaixin Sui, Xiping Wang, Dapeng Liu, Xing Jin, Qi Wang, and Dan Pei. 2019. Generic and robust localization of multi-dimensional root causes. In *Proceedings of the IEEE 30th International Symposium on Software Reliability Engineering (ISSRE'19)*. IEEE, IEEE, 47–57.

[81] Yinglung Liang, Yanyong Zhang, Hui Xiong, and Ramendra Sahoo. 2007. Failure prediction in IBM bluegene/l event logs. In *Proceedings of the 7th IEEE International Conference on Data Mining (ICDM'07)*. IEEE, 583–588. https://doi.org/10.1109/icdm.2007.46

[82] Fan Lin, Matt Beadon, Harish Dattatraya Dixit, Gautham Vunnam, Amol Desai, and Sriram Sankar. 2018. Hardware remediation at scale. In *Proceedings of the 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W'18)*. IEEE, 14–17. https://doi.org/10.1109/dsn-w.2018.00015

[83] Fred Lin, Keyur Muzumdar, Nikolay Pavlovich Laptev, Mihai-Valentin Curelea, Seunghak Lee, and Sriram Sankar. 2020. Fast dimensional analysis for root cause investigation in a large-scale service environment. *Proc. ACM Meas. Anal. Comput. Syst.* 4, 2, Article 31 (June 2020), 23 pages. https://doi.org/10.1145/3392149

[84] Qingwei Lin, Hongyu Zhang, Jian-Guang Lou, Yu Zhang, and Xuewei Chen. 2016. Log clustering based problem identification for online service systems. In *Proceedings of the 38th International Conference on Software Engineering Companion (ICSE'16)*. ACM, New York, NY, 102–111. https://doi.org/10.1145/2889160.2889232

[85] Chao Liu, Xifeng Yan, Long Fei, Jiawei Han, and Samuel P. Midkiff. 2005. SOBER: Statistical model-based bug localization. *ACM SIGSOFT Softw. Eng. Notes* 30, 5 (Sep. 2005), 286. https://doi.org/10.1145/1095430.1081753

[86] Dapeng Liu, Youjian Zhao, Haowen Xu, Yongqian Sun, Dan Pei, Jiao Luo, Xiaowei Jing, and Mei Feng. 2015. Opprentice: Towards practical and automatic anomaly detection through machine learning. In *Proceedings of the Internet Measurement Conference*. ACM, New York, NY, 211–224. https://doi.org/10.1145/2815675.2815679

[87] David Lo, Hong Cheng, Jiawei Han, Siau-Cheng Khoo, and Chengnian Sun. 2009. Classification of software behaviors for failure detection: A discriminative pattern mining approach. In *Proceedings of the 15th ACM SIGKDD International*

Conference on Knowledge Discovery and Data Mining (KDD'09). ACM, New York, NY, 557–566. https://doi.org/10.1145/1557019.1557083

[88] Ao Ma, Fred Douglis, Guanlin Lu, Darren Sawyer, Surendar Chandra, and Windsor Hsu. 2015. RAIDShield: Characterizing, monitoring, and proactively protecting against disk failures. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*. USENIX Association, 241–256. https://www.usenix.org/conference/fast15/technical-sessions/presentation/ma

[89] Farzaneh Mahdisoltani, Ioan Stefanovici, and Bianca Schroeder. 2017. Proactive error prediction to improve storage system reliability. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'17)*. USENIX Association, 391–402. Retrieved from https://www.usenix.org/conference/atc17/technical-sessions/presentation/mahdisoltani.

[90] T. J. McCabe. 1976. A complexity measure. *IEEE Trans. Softw. Eng.* SE-2, 4 (Dec. 1976), 308–320. https://doi.org/10.1109/TSE.1976.233837

[91] Meiliana, Syaeful Karim, Harco Leslie Hendric Spits Warnars, Ford Lumban Gaol, Edi Abdurachman, and Benfano Soewito. 2017. Software metrics for fault prediction using machine learning approaches: A literature review with PROMISE repository dataset. In *Proceedings of the IEEE International Conference on Cybernetics and Computational Intelligence (CyberneticsCom'17)*. IEEE, 19–23. https://doi.org/10.1109/cyberneticscom.2017.8311708

[92] Weibin Meng, Ying Liu, Yichen Zhu, Shenglin Zhang, Dan Pei, Yuqing Liu, Yihao Chen, Ruizhi Zhang, Shimin Tao, Pei Sun, et al. 2019. LogAnomaly: Unsupervised detection of sequential and quantitative anomalies in unstructured logs. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI'19)*. 4739–4745. https://doi.org/10.24963/ijcai.2019/658

[93] Tim Menzies. 2004. PROMISE DATASETS PAGE. Retrieved from http://promise.site.uottawa.ca/SERepository/datasets-page.html.

[94] Tim Menzies, Jeremy Greenwald, and Art Frank. 2007. Data mining static code attributes to learn defect predictors. *IEEE Trans. Softw. Eng.* 33, 1 (Jan. 2007), 2–13. https://doi.org/10.1109/TSE.2007.256941

[95] Adam Moody, Greg Bronevetsky, Kathryn Mohror, and de Bronis R. Supinski. 2010. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–11. https://doi.org/10.1109/sc.2010.18

[96] Moogsoft. 2020. What Is AIOps? Moogsoft. Retrieved from https://www.moogsoft.com/resources/aiops/guide/everything-aiops/.

[97] Andrew W. Moore and Denis Zuev. 2005. Internet traffic classification using bayesian analysis techniques. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'05)*. ACM, New York, NY, 50–60. https://doi.org/10.1145/1064212.1064220

[98] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. 2008. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the 13th International Conference on Software Engineering*. ACM, New York, NY, 181–190. https://doi.org/10.1145/1368088.1368114

[99] Mukosi Abraham Mukwevho and Turgay Celik. 2021. Toward a smart cloud: A review of fault-tolerance methods in cloud systems. *IEEE Trans. Serv. Comput.* 14, 2 (2021), 589–605. https://doi.org/10.1109/tsc.2018.2816644

[100] Joseph F. Murray, Gordon F. Hughes, and Kenneth Kreutz-Delgado. 2003. Hard drive failure prediction using non-parametric statistical methods. Retrieved from http://dsp.ucsd.edu/~jfmurray/publications/Murray2003.pdf.

[101] Joseph F. Murray, Gordon F. Hughes, and Kenneth Kreutz-Delgado. 2005. Machine learning methods for predicting failures in hard drives: A multiple-instance application. *J. Mach. Learn. Res.* 6 (Jan. 2005), 783–816. https://doi.org/10.5555/1046920.1088699

[102] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. 2006. Mining metrics to predict component failures. In *Proceeding of the 28th International Conference on Software Engineering (ICSE'06)*. ACM, New York, NY, 452–461. https://doi.org/10.1145/1134285.1134349

[103] Jaechang Nam, Sinno Jialin Pan, and Sunghun Kim. 2013. Transfer defect learning. In *Proceedings of the 35th International Conference on Software Engineering (ICSE'13)*. IEEE, 382–391. https://doi.org/10.1109/icse.2013.6606584

[104] Iyswarya Narayanan, Kushagra Vaid, Di Wang, Myeongjae Jeon, Bikash Sharma, Laura Caulfield, Anand Sivasubramaniam, Ben Cutler, Jie Liu, and Badriddine Khessib. 2016. SSD failures in datacenters: What? When? and Why? In *Proceedings of the 9th ACM International on Systems and Storage Conference (SYSTOR'16)*. ACM, New York, NY, Article 7, 11 pages. https://doi.org/10.1145/2928275.2928278

[105] Roberto Natella, Domenico Cotroneo, Joao A. Duraes, and Henrique S. Madeira. 2013. On fault representativeness of software fault injection. *IEEE Trans. Softw. Eng.* 39, 1 (Jan. 2013), 80–96. https://doi.org/10.1109/tse.2011.124

[106] Roberto Natella, Domenico Cotroneo, and Henrique S. Madeira. 2016. Assessing dependability with software fault injection: A survey. *Comput. Surveys* 48, 3 (Aug. 2016), 44:1–44:55. https://doi.org/10.1145/2841425

[107] Sasho Nedelkoski, Jorge Cardoso, and Odej Kao. 2019. Anomaly detection and classification using distributed tracing and deep learning. In *Proceedings of the 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID'19)*. IEEE, 241–250. https://doi.org/10.1109/ccgrid.2019.00038

[108] NetManAIOps. 2019. SMD Dataset—OmniAnomaly. Retrieved from https://github.com/NetManAIOps/OmniAnomaly.
[109] Clodoaldo Brasilino Leite Neto, Pedro Batista De Carvalho Filho, and Alexandre NÃşbrega Duarte. 2013. A systematic mapping study on fault management in cloud computing. In *Proceedings of the International Conference on Parallel and Distributed Computing, Applications and Technologies*. IEEE, 332–337. https://doi.org/10.1109/PDCAT.2013.59
[110] Hiep Nguyen, Zhiming Shen, Yongmin Tan, and Xiaohui Gu. 2013. FChain: Toward black-box online fault localization for cloud systems. In *Proceedings of the IEEE 33rd International Conference on Distributed Computing Systems*. IEEE, 21–30. https://doi.org/10.1109/icdcs.2013.26
[111] Thuy T. T. Nguyen and Grenville Armitage. 2008. A survey of techniques for internet traffic classification using machine learning. *IEEE Commun. Surveys Tutor.* 10, 4 (2008), 56–76. https://doi.org/10.1109/surv.2008.080406
[112] Changhai Nie and Hareton Leung. 2011. A survey of combinatorial testing. *Comput. Surveys* 43, 2 (Apr. 2011), 11:1–11:29. https://doi.org/10.1145/1883612.1883618
[113] Paolo Notaro, Jorge Cardoso, and Michael Gerndt. 2020. A systematic mapping study in AIOps. In *Proceedings of the International Conference on Service-oriented Computing (ICSOC'20). Workshops: AIOps, CFTIC, STRAPS, AI-PA, AI-IOTS, and Satellite Events*. Springer, 110–123. Retrieved from http://arxiv.org/abs/2012.09108.
[114] H. Okamura, Y. Nishimura, and T. Dohi. 2004. A dynamic checkpointing scheme based on reinforcement learning. In *Proceedings of the 10th IEEE Pacific Rim International Symposium on Dependable Computing*. IEEE, 151–158. https://doi.org/10.1109/PRDC.2004.1276566
[115] Ahmet Okutan and Olcay Taner Yıldız. 2012. Software defect prediction using bayesian networks. *Empir. Softw. Eng.* 19, 1 (Aug. 2012), 154–181. https://doi.org/10.1007/s10664-012-9218-8
[116] OpsRamp. 2020. AIOps (AI for IT Operations)—OpsRamp. Retrieved from https://www.opsramp.com/solutions/service-centric-aiops/.
[117] T. J. Ostrand, E. J. Weyuker, and R. M. Bell. 2005. Predicting the location and number of faults in large software systems. *IEEE Trans. Softw. Eng.* 31, 4 (Apr. 2005), 340–355. https://doi.org/10.1109/tse.2005.49
[118] Eduardo Pinheiro, Wolf-Dietrich Weber, and Luiz André Barroso. 2007. Failure trends in a large disk drive population. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST'07)*. USENIX Association, 2. Retrieved from https://www.usenix.org/conference/fast-07/failure-trends-large-disk-drive-population.
[119] Teerat Pitakrat, Dušan Okanović, André van Hoorn, and Lars Grunske. 2018. Hora: Architecture-aware online failure prediction. *J. Syst. Softw.* 137 (Mar. 2018), 669–685. https://doi.org/10.1016/j.jss.2017.02.041
[120] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, Jiayang Sun, and Bin Wang. 2003. Automated support for classifying software failure reports. In *Proceedings of the 25th International Conference on Software Engineering*. IEEE, 465–475. https://doi.org/10.1109/icse.2003.1201224
[121] M. Renieris and S. P. Reiss. 2003. Fault localization with nearest neighbor queries. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering*. IEEE, 30–39. https://ieeexplore.ieee.org/document/1240292
[122] Felix Salfner, Maren Lenk, and Miroslaw Malek. 2010. A survey of online failure prediction methods. *Comput. Surveys* 42, 3 (Mar. 2010), 1–42. https://doi.org/10.1145/1670679.1670680
[123] Felix Salfner and Miroslaw Malek. 2007. Using hidden semi-Markov models for effective online failure prediction. In *Proceedings of the 26th IEEE International Symposium on Reliable Distributed Systems (SRDS'07)*. IEEE, 161–174. https://doi.org/10.1109/srds.2007.35
[124] Areeg Samir and Claus Pahl. 2019. A Controller Architecture for Anomaly Detection, Root Cause Analysis and Self-Adaptation for Cluster Architectures. Retrieved from https://orbilu.uni.lu/handle/10993/42062.
[125] Mark Schwabacher and Kai Goebel. 2007. A survey of artificial intelligence for prognostics. In *Proceedings of the AAAI Fall Symposium on Artificial Intelligence for Prognostics*. AAAI, 108–115. Retrieved from https://www.aaai.org/Library/Symposia/Fall/2007/fs07-02-016.php.
[126] Qihong Shao, Yi Chen, Shu Tao, Xifeng Yan, and Nikos Anerousis. 2008. Efficient ticket routing by resolution sequence mining. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'08)*. ACM, New York, NY, 605–613. https://doi.org/10.1145/1401890.1401964
[127] Bikash Sharma, Praveen Jayachandran, Akshat Verma, and Chita R. Das. 2013. CloudPD: Problem determination and diagnosis in shared dynamic clouds. In *Proceedings of the 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'13)*. IEEE, 1–12. https://doi.org/10.1109/dsn.2013.6575298
[128] Akbar Siami Namin, James H. Andrews, and Duncan J. Murdoch. 2008. Sufficient mutation operators for measuring test effectiveness. In *Proceedings of the 13th International Conference on Software Engineering (ICSE'08)*. ACM, New York, NY, 351–360. https://doi.org/10.1145/1368088.1368136
[129] BMC Software. 2020. AIOps—BMC. Retrieved from https://www.bmc.com/it-solutions/aiops.html.
[130] Marc Solé, Victor Muntés-Mulero, Annie Ibrahim Rana, and Giovani Estrada. 2017. Survey on Models and Techniques for Root-Cause Analysis. Retrieved from http://arxiv.org/abs/1701.08546.

[131] Ya Su, Youjian Zhao, Chenhao Niu, Rong Liu, Wei Sun, and Dan Pei. 2019. Robust anomaly detection for multivariate time series through stochastic recurrent neural network. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'19)*. ACM, New York, NY, 2828–2837.

[132] Yongqian Sun, Youjian Zhao, Ya Su, Dapeng Liu, Xiaohui Nie, Yuan Meng, Shiwen Cheng, Dan Pei, Shenglin Zhang, Xianping Qu et al. 2018. Hotspot: Anomaly localization for additive kpis with multi-dimensional attributes. *IEEE Access* 6 (2018), 10909–10923.

[133] Resolve Systems. 2020. What is AIOps?—Resolve. Retrieved from https://resolve.io/what-is-aiops.

[134] D. Tang and R. K. Iyer. 1993. Dependability measurement and modeling of a multicomputer system. *IEEE Trans. Comput.* 42, 1 (1993), 62–75. https://doi.org/10.1109/12.192214

[135] Timothy K. Tsai and Ravishankar K. Iyer. 1995. FTAPE: A fault injection tool to measure fault Tolerance. NASA STI/Recon Technical Report. 25333 pages. https://doi.org/10.2514/6.1995-1041

[136] K. Vaidyanathan and K. S. Trivedi. 1999. A measurement-based model for estimation of resource exhaustion in operational software systems. In *Proceedings of the 10th International Symposium on Software Reliability Engineering*. IEEE, 84–93. https://doi.org/10.1109/issre.1999.809313

[137] K. Vaidyanathan and K. S. Trivedi. 2005. A comprehensive model for software rejuvenation. *IEEE Trans. Depend. Secure Comput.* 2, 2 (Feb. 2005), 124–137. https://doi.org/10.1109/tdsc.2005.15

[138] Kashi Venkatesh Vishwanath and Nachiappan Nagappan. 2010. Characterizing cloud computing hardware reliability. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC'10)*. ACM, New York, NY, 193–204. https://doi.org/10.1145/1807128.1807161

[139] Jing Wang, Daniel Rossell, Christos G. Cassandras, and Ioannis Ch. Paschalidis. 2013. Network anomaly detection: A survey and comparative analysis of stochastic and deterministic methods. In *Proceedings of the 52nd IEEE Conference on Decision and Control*. IEEE, 182–187. https://doi.org/10.1109/CDC.2013.6759879

[140] Qing Wang, Wubai Zhou, Chunqiu Zeng, Tao Li, Larisa Shwartz, and Genady Ya. Grabarnik. 2017. Constructing the knowledge base for cognitive IT service management. In *Proceedings of the IEEE International Conference on Services Computing (SCC'17)*. IEEE, 410–417. https://doi.org/10.1109/scc.2017.59

[141] Song Wang, Taiyue Liu, and Lin Tan. 2016. Automatically learning semantic features for defect prediction. In *Proceedings of the 38th International Conference on Software Engineering (ICSE'16)*. ACM, New York, NY, 297–308. https://doi.org/10.1145/2884781.2884804

[142] Wei Wang, Ming Zhu, Jinlin Wang, Xuewen Zeng, and Zhongzhen Yang. 2017. End-to-end encrypted traffic classification with one-dimensional convolution neural networks. In *Proceedings of the IEEE International Conference on Intelligence and Security Informatics (ISI'17)*. IEEE, 43–48. https://doi.org/10.1109/isi.2017.8004872

[143] Yu Wang, Qiang Miao, Eden W. M. Ma, Kwok-Leung Tsui, and Michael G. Pecht. 2013. Online anomaly detection for hard disk drives based on mahalanobis distance. *IEEE Trans. Reliabil.* 62, 1 (Mar. 2013), 136–145. https://doi.org/10.1109/tr.2013.2241204

[144] Amy Ward, Peter Glynn, and Kathy Richardson. 1998. Internet service performance failure detection. *ACM SIGMETRICS Perform. Eval. Rev.* 26, 3 (Dec. 1998), 38–43. https://doi.org/10.1145/306225.306237

[145] W. Eric Wong, Vidroha Debroy, Ruizhi Gao, and Yihao Li. 2014. The DStar method for effective software fault localization. *IEEE Trans. Reliabil.* 63, 1 (Mar. 2014), 290–308. https://doi.org/10.1109/tr.2013.2285319

[146] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A survey on software fault localization. *IEEE Trans. Softw. Eng.* 42, 8 (Aug. 2016), 707–740. https://doi.org/10.1109/tse.2016.2521368

[147] Rongxin Wu, Hongyu Zhang, Sunghun Kim, and Shing-Chi Cheung. 2011. ReLink: Recovering links between bugs and changes. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE'11)*. ACM, 15–25. https://doi.org/10.1145/2025113.2025120

[148] Jiang Xiao, Zhuang Xiong, Song Wu, Yusheng Yi, Hai Jin, and Kan Hu. 2018. Disk failure prediction in data centers via online learning. In *Proceedings of the 47th International Conference on Parallel Processing (ICPP'18)*. ACM, New York, NY, Article 35, 10 pages. https://doi.org/10.1145/3225058.3225106

[149] Chang Xu, Gang Wang, Xiaoguang Liu, Dongdong Guo, and Tie-Yan Liu. 2016. Health status assessment and failure prediction for hard drives with recurrent neural networks. *IEEE Trans. Comput.* 65, 11 (Nov. 2016), 3502–3508. https://doi.org/10.1109/tc.2016.2538237

[150] Haowen Xu, Yang Feng, Jie Chen, Zhaogang Wang, Honglin Qiao, Wenxiao Chen, Nengwen Zhao, Zeyan Li, Jiahao Bu, Zhihan Li et al. 2018. Unsupervised anomaly detection via variational auto-encoder for seasonal KPIs in web applications. In *Proceedings of the World Wide Web Conference (WWW'18)*. ACM, New York, NY, 187–196. https://doi.org/10.1145/3178876.3185996

[151] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I. Jordan. 2009. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP'09)* . ACM, New York, NY, 117–132. https://doi.org/10.1145/1629575.1629587

[152] Zhenghua Xue, Xiaoshe Dong, Siyuan Ma, and Weiqing Dong. 2007. A survey on failure prediction of large-scale server clusters. In *Proceedings of the 8th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD'07)*. IEEE, 733–738. https://doi.org/10.1109/snpd.2007.284

[153] Xiaoxing Yang, Ke Tang, and Xin Yao. 2015. A learning-to-rank approach to software defect prediction. *IEEE Trans. Reliabil.* 64, 1 (3 2015), 234–246. https://doi.org/10.1109/tr.2014.2370891

[154] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. 2010. SherLog: Error diagnosis by connecting clues from run-time logs. *ACM SIGARCH Comput. Architect. News* 38, 1 (Mar. 2010), 143–154. https://doi.org/10.1145/1735970.1736038

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

[156] Andreas Zeller. 2002. Isolating cause-effect chains from computer programs. In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering (SIGSOFT'02/FSE'10)*. ACM, New York, NY, 1–10. https://doi.org/10.1145/587051.587053

[157] Andreas Zeller. 2006. Eclipse Bug Data!—Software Engineering Chair (Prof. Zeller)—Saarland University. Retrieved from https://www.st.cs.uni-saarland.de/softevo/bug-data/eclipse/.

[158] Chunqiu Zeng, Wubai Zhou, Tao Li, Larisa Shwartz, and Genady Ya Grabarnik. 2017. Knowledge guided hierarchical multi-label classification over ticket data. *IEEE Trans. Netw. Service Manage.* 14, 2 (6 2017), 246–260. https://doi.org/10.1109/tnsm.2017.2668363

[159] Chuxu Zhang, Dongjin Song, Yuncong Chen, Xinyang Feng, Cristian Lumezanu, Wei Cheng, Jingchao Ni, Bo Zong, Haifeng Chen, and Nitesh V. Chawla. 2019. A deep neural network for unsupervised anomaly detection and diagnosis in multivariate time series data. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. AAAI, 1409–1416.

[160] Ke Zhang, Jianwu Xu, Martin Renqiang Min, Guofei Jiang, Konstantinos Pelechrinis, and Hui Zhang. 2016. Automated IT system failure prediction: A deep learning approach. In *Proceedings of the IEEE International Conference on Big Data (BigData'16)*. IEEE, 1291–1300. https://doi.org/10.1109/bigdata.2016.7840733

[161] Shenglin Zhang, Weibin Meng, Jiahao Bu, Sen Yang, Ying Liu, Dan Pei, Jun Xu, Yu Chen, Hui Dong, Xianping Qu, and et al. 2017. Syslog processing for switch failure diagnosis and prediction in datacenter networks. In *Proceedings of the IEEE/ACM 25th International Symposium on Quality of Service (IWQoS'17)*. IEEE, 1–10. https://doi.org/10.1109/iwqos.2017.7969130

[162] Xu Zhang, Yong Xu, Qingwei Lin, Bo Qiao, Hongyu Zhang, Yingnong Dang, Chunyu Xie, Xinsheng Yang, Qian Cheng, Ze Li, Junjie Chen, Xiaoting He, Randolph Yao, Jian-Guang Lou, Murali Chintalapati, Furao Shen, and Dongmei Zhang. 2019. Robust log-based anomaly detection on unstable log data. In *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'19)*. ACM, New York, NY, 807–817. https://doi.org/10.1145/3338906.3338931

[163] Xu Zhao, Kirk Rodrigues, Yu Luo, Michael Stumm, Ding Yuan, and Yuanyuan Zhou. 2017. Log20: Fully automated optimal placement of log printing statements under specified overhead threshold. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17)*. ACM, New York, NY, 565–581. https://doi.org/10.1145/3132747.3132778

[164] Ying Zhao, Xiang Liu, Siqing Gan, and Weimin Zheng. 2010. Predicting disk failures with HMM- and HSMM-based approaches. In *Proceedings of the 10th Industrial Conference on Advances in Data Mining: Applications and Theoretical Aspects (ICDM'10)*. Springer-Verlag, Berlin, 390–404.

[165] Shuai Zheng, Kosta Ristovski, Ahmed Farahat, and Chetan Gupta. 2017. Long short-term memory network for remaining useful life estimation. In *Proceedings of the IEEE International Conference on Prognostics and Health Management (ICPHM'17)* (2017–06). IEEE, 88–95. https://doi.org/10.1109/icphm.2017.7998311

[166] Wubai Zhou, Liang Tang, Tao Li, Larisa Shwartz, and Genady Ya. Grabarnik. 2015. Resolution recommendation for event tickets in service management. In *Proceedings of the IFIP/IEEE International Symposium on Integrated Network Management (IM'15)*. IEEE, 287–295. https://doi.org/10.1109/inm.2015.7140303

[167] Bingpeng Zhu, Gang Wang, Xiaoguang Liu, Dianming Hu, Sheng Lin, and Jingwei Ma. 2013. Proactive drive failure prediction for large scale storage systems. In *Proceedings of the IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST'13)*. IEEE, 1–5. https://doi.org/10.1109/msst.2013.6558427

[168] Jieming Zhu, Pinjia He, Qiang Fu, Hongyu Zhang, Michael R. Lyu, and Dongmei Zhang. 2015. Learning to log: Helping developers make informed logging decisions. In *Proceedings of the IEEE/ACM 37th IEEE International Conference on Software Engineering*. IEEE, 415–425. https://doi.org/10.1109/icse.2015.60

Received April 2021; revised July 2021; accepted August 2021