USING UNSUPERVISED MACHINE LEARNING FOR FAULT IDENTIFICATION IN VIRTUAL MACHINES

Chris Schneider

A Thesis Submitted for the Degree of PhD
at the
University of St Andrews

2015

Using Unsupervised Machine Learning for Fault Identification in Virtual Machines

Chris Schneider

This thesis is submitted in partial fulfillment for the degree of

Doctor of Philosophy

at the University of St Andrews

June 2015

Abstract

Self-healing systems promise operating cost reductions in large-scale computing environments through the automated detection of, and recovery from, faults. However, at present appears to be little known empirical evidence comparing the different approaches, or demonstrations that such implementations reduce costs.

This thesis compares previous and current self-healing approaches before demonstrating a new, unsupervised approach that combines artificial neural networks with performance tests to perform fault identification in an automated fashion, *i.e.* the correct and accurate determination of which computer features are associated with a given performance test failure.

Several key contributions are made in the course of this research including an analysis of the different types of self-healing approaches based on their contextual use, a baseline for future comparisons between self-healing frameworks that use artificial neural networks, and a successful, automated fault identification in cloud infrastructure, and more specifically virtual machines. This approach uses three established machine learning techniques: Naïve Bayes, Baum-Welch, and Contrastive Divergence Learning. The latter demonstrates minimisation of human-interaction beyond previous implementations by producing a list in decreasing order of likelihood of potential root causes (*i.e.* fault hypotheses) which brings the state of the art one step closer toward fully self-healing systems.

This thesis also examines the impact of that different types of faults have on their respective identification. This helps to understand the validity of the data being presented, and how the field is progressing, whilst examining the differences in impact to identification between emulated thread crashes and errant user changes – a contribution believed to be unique to this research.

Lastly, future research avenues and conclusions in automated fault identification are described along with lessons learned throughout this endeavor. This includes the progression of artificial neural networks, how learning algorithms are being developed and understood, and possibilities for automatically generating feature locality data.

Acknowledgements

This research has been primarily funded by The Scottish Informatics and Computer Science Alliance (SICSA) and by the University of St Andrews, and made possible via the generous time and personal investments of other academics. Thanks go to the following people, in particular:

My supervisors Simon Dobson and Adam Barker, along with Saleem Bhatti and Graham Kirby – all of whom faculty members of The School of Computer Science at the University of St Andrews – and my colleagues Ruth Hoffman and Ildikó Pete – for their guidance, recommendations, time spent proofing, and their invaluable feedback.

Brant Moriarity, Larry Yaeger, Mehmet Dalkilic, Luis Rocha, and Marty Siegel of Indiana University for lessons both academic and social, and their personal investments in my success.

Matt and Brea Carlson, most recently of Wabash College but also of Indiana University, for their guidance, encouragement, and recommendations in getting me started with the academy, and for their continued support.

Susan Hohenberger (Waters) of The Johns Hopkins University for their investment in my education and furthering my foundational knowledge in computer science.

And, to Jeff House, who taught me to program in his own free time and asked nothing in return.

Without the involvement and generosity of these individuals and organisations, it is easy to imagine a life of less success. My fealty, and kindest, deepest thanks to you all.

1. Candidate's declarations:

I, Chris Schneider, hereby certify that this thesis, which is approximately 29,000 words in length, has been written by me, and that it is the record of work carried out by me and that it has not been submitted in any previous application for a higher degree.

I was admitted as a research student in Sep. 2011 and as a candidate for the degree of PhD in [month, year]; the higher study for which this is a record was carried out in the University of St Andrews between 2011 and 2015.

Date 5 Aug. 2015 signature of candidate

2. Supervisor's declaration:

I hereby certify that the candidate has fulfilled the conditions of the Resolution and Regulations appropriate for the degree of PhD in the University of St Andrews and that the candidate is qualified to submit this thesis in application for that degree.

Date 5 Aug. 2015 signature of supervisor

3. Permission for electronic publication: *(to be signed by both candidate and supervisor)*

In submitting this thesis to the University of St Andrews we understand that we are giving permission for it to be made available for use in accordance with the regulations of the University Library for the time being in force, subject to any copyright vested in the work not being affected thereby. We also understand that the title and the abstract will be published, and that a copy of the work may be made and supplied to any bona fide library or research worker, that my thesis will be electronically accessible for personal or research use unless exempt by award of an embargo as requested below, and that the library has the right to migrate my thesis into new electronic forms as required to ensure continued access to the thesis. We have obtained any third-party copyright permissions that may be required in order to allow such access and migration, or have requested the appropriate embargo below.

The following is an agreed request by candidate and supervisor regarding the publication of this thesis:

PRINTED COPY

a) No embargo on print copy

~~b) Embargo on all or part of print copy for a period of … years (maximum five years) on the following ground(s):~~

- ~~Publication would be commercially damaging to the researcher, or to the supervisor, or the University~~
- ~~Publication would preclude future publication~~
- ~~Publication would be in breach of laws or ethics~~

~~c) Permanent or longer term embargo on all or part of print copy for a period of … years (the request will be referred to the Pro-Provost and permission will be granted only in exceptional circumstances).~~

~~**Supporting statement for printed embargo request:**~~

ELECTRONIC COPY

a) No embargo on electronic copy

~~b) Embargo on all or part of electronic copy for a period of … years (maximum five years) on the following ground(s):~~

- ~~Publication would be commercially damaging to the researcher, or to the supervisor, or the University~~
- ~~Publication would preclude future publication~~
- ~~Publication would be in breach of law or ethics~~

~~c) Permanent or longer term embargo on all or part of electronic copy for a period of … years (the request will be referred to the Pro-Provost and permission will be granted only in exceptional circumstances).~~

~~**Supporting statement for electronic embargo request:**~~

Date 5 Aug. 2015 signature of candidate signature of supervisor

Please note initial embargoes can be requested for a maximum of five years. An embargo on a thesis submitted to the Faculty of Science and Medicine is rarely granted for more than two years in the first instance, without good justification. The Library will not lift an embargo before confirming with the student and supervisor that they do not intend to request a continuation. In the absence of an agreed response from both student and supervisor, the Head of School will be consulted. Please note that the total period of an embargo, including a continuation, is not expected to exceed ten years.
Where part of a thesis is to be embargoed, please specify the part and the reason.

For Emma, who always believed.

Thank you.

CONTENTS

Contents i

List of Figures iii

List of Tables vii

1 Introduction 5
- 1.1 Motivation . . . 6
- 1.2 Hypothesis & Central Tenets . . . 7
  - 1.2.1 Hypothesis . . . 7
  - 1.2.2 Central Claims . . . 8
- 1.3 Main Contributions . . . 8
- 1.4 Definitions . . . 9
- 1.5 Published Works . . . 12
- 1.6 Organisation . . . 14

2 Background 15
- 2.1 Introduction . . . 15
  - 2.1.1 Terminology . . . 16
  - 2.1.2 Assumptions . . . 18
- 2.2 The History of Autonomic Computing . . . 20
  - 2.2.1 Self-* Systems . . . 21
  - 2.2.2 Self-Healing Systems . . . 23
- 2.3 Machine Learning Techniques . . . 24
  - 2.3.1 Artificial Neural Networks . . . 25
  - 2.3.2 Hidden Markov Models . . . 26
  - 2.3.3 Restricted Boltzmann Machines . . . 27
  - 2.3.4 Stochastic Primitives . . . 29

3 A Systematic Review of Self-Healing Systems 31
- 3.1 Methodology . . . 31
  - 3.1.1 Search Process . . . 32
  - 3.1.2 Research Questions . . . 33
  - 3.1.3 Quality Assessment . . . 34
  - 3.1.4 Data Collection & Analysis . . . 34
  - 3.1.5 Results . . . 35

3.2 A Comparison of Self-Healing Systems . . . 35
3.2.1 Management Styles . . . 36
3.2.2 Computing Environments . . . 44
3.2.3 Learning Methodologies . . . 49
3.3 Synthesis . . . 54
3.4 Synopsis . . . 58

4 An Automated Approach for Identifying Faults 63
4.1 Problem Description . . . 63
4.2 Approach . . . 64
4.2.1 Running Example . . . 67
4.3 Experiments . . . 68
4.3.1 Hidden Markov Models & Artificial Neural Networks . . . 70
4.3.2 Restricted Boltzmann Machines . . . 72
4.4 Limitations . . . 73
4.5 Threats to Validity . . . 75
4.5.1 Construct . . . 75
4.5.2 Internal . . . 76
4.5.3 External . . . 77
4.6 Implementation . . . 78
4.7 Comparison & Inference . . . 84
4.7.1 Baseline Establishment . . . 84
4.7.2 UBL - An External Basis for Comparison . . . 85
4.7.3 Collection . . . 88
4.7.4 Classification . . . 89
4.7.5 Learning & Analysis . . . 91
4.7.6 Comparison Constraints . . . 96

5 Results & Discussion 99
5.1 Introduction . . . 99
5.2 Overview . . . 100
5.3 Results . . . 101
5.3.1 The FDFs . . . 103
5.3.2 UBL . . . 116
5.4 Discussion . . . 119

6 Conclusion 127
6.1 Findings . . . 127
6.2 Lessons . . . 128
6.3 Future Work . . . 131

Appendix A Appendix-A 133
A.1 UBL Results . . . 133

References 135

List of Figures

2.1 **Artificial Neural Network.** ANNs are a type of statistical model that operates by updating weights along paths between hidden and visible layers to forecast or otherwise 'learn' a series of inputs. 25

2.2 **Hidden Markov Model.** HMMs operate by forecasting the hidden layer (Zs) using observations from a visible set of inputs (*i.e.* a Markov chain, Xs). Unlike ANNs, HMMs do not use more than two independent layers to separate observed and hidden data. 27

2.3 **Restricted Boltzmann Machine.** RBMs operate similarly to ANNs and HMMs, but adjust their weights by using an form of alternating Gibbs sampling. This allows them to update their layers in parallel – an advantage over other stochastic primitives. 29

3.1 **Management Styles versus Computing Environments.** Managed environments – such as Cloud and n-Tier infrastructures – show a preference for top-down management styles, whereas ad hoc computing environments prefer bottom-up management styles. 43

3.2 **Learning Methodologies versus Computing Environments.** Most self-healing systems prefer a supervised learning methodology, regardless of the environment it is implemented in. This is useful for ensuring correct behaviours, but also a limitation in potential for autonomy. 48

3.3 **Learning Methodologies versus Management Styles.** When self-healing system are implemented in a top-down fashion, they tend to leverage supervised learning methodologies. Likewise, bottom-up management styles are more likely to use unsupervised and semi-supervised learning. 54

3.4 **Self-Healing Systems Frameworks.** Self-healing systems frameworks as categorised by learning methodology, computing environment, and expected management style by first author's last name, year of introduction, and framework title (if appropriate). In some cases frameworks exhibit abilities to operate under multiple assumptions – these incidents are represented by additional bullets within the graph. Figure 2.8 divides this information into percentages with each entry represented once per category. 55

3.5 **Relative Coverage of Different Self-Healing Techniques.** 56

4.1 **Fault Detection Framework Logic & Architecture Diagram using Greedy Ingest.** The FDF leveraging ANNs and HMMs operates by updating its primitives as soon as feature data is recovered from the system. 66

4.2 **Fault Detection Framework Logic & Architecture Diagram using Lazy Ingest.** The FDF leveraging RBMs operates identically to the FDF that uses ANNs and HMMs except with a lazy ingest mechanism for feature behaviour data. Primitives using a lazy ingest are only trained upon fault detection. 67
4.3 **Running Example – Successful Data Collection via FDFs.** Cropped image showing successful data collection when running an RBM-based FDF. 68
4.4 **Running Example – Fault Identification via FDFs.** Cropped image showing a sample of an FDF result screen. The full sized list has been truncated to save space but can contain 5 to 300 leads. 68
4.5 **Fault Detection Framework Logic & Architecture using Hidden Markov Models and Artificial Neural Networks.** Fault Detection Frameworks are provided three inputs, set to run, and then injected with faults at varying time intervals. The result is an ordered list of leads based on forecasted feature behaviours. 71

5.1 **FDF v1.0 - Time Taken.** Time-Taken represents the number of "*ElapsedTicks*" converted to milliseconds (ms) between when a fault is detected and the return of an ordered list of potential root causes. The ANN took less time than the HMM to produce an ordered list of fault hypotheses. Shortened times allow for a wider range of recovery solutions making them more desirable. Both values grow linearly per the amount of data being provided. 105
5.2 **FDF v1.0 - Confidence.** Confidence conveys how likely the FDF's suspect a given lead is associated with the correct root cause of the detected fault. Converse to the amount of time taken, the HMM produced much higher confidence values than the ANN. This is a result that was unexpected because of the way that the Baum-Welch algorithm calculates probabilities. 105
5.3 **FDF v1.0 - Fault Position.** The average position of a correct root cause as returned by the FDF is represented in this graph. As the lists are ordered by descending probability, lower values are better. The averages from the experiments show that the HMM outperforms the ANN in nearly every test. Additionally, fault position improvement is much slower with the ANN. 107
5.4 **FDF v1.0 - Total Leads.** FDFs generate leads when a fault is detected. This graph represents the average total number of suspect features (*i.e.* 'leads') at 5-point sample intervals. The FDF using HMMs is able to generate more leads than the one using ANNs, however more leads is not always better. The ideal result is a list containing only the features that are associated with the cause of the fault. 107
5.5 **FDF v1.0 - Precision.** HMMs provide more precise results initially, but eventually trade with ANNs. This is significant in that neither approach is particularly precise, but as more information is added, the HMM appears to drop in precision. This result correlates with more leads being generated, and some second ordering issues. 109
5.6 **FDF v1.0 - F-Measure.** The F-Measure represents the overall performance of the learning algorithms in relation to both Baum-Welch (HMM), and Naïve Bayes (ANN) in terms of precision whilst accounting for outliers. Due to the way the trials were executed, similar results were obtained in each examination. 109

5.7 **FDF v2.0 - Time Taken.** A switch from greedy to lazy data ingest caused an expected increased time-based performance metrics. This is because all of the calculations for training the RBMs was performed once a fault was detected versus after every collection sample. 111
5.8 **FDF v2.0 - Confidence.** The gradual increase in confidence values via the CDL appears to be more robust than previous approaches. Rather than a relative value being given or steep increases, a gradual pattern emerges as more data is fed into the FDF. 111
5.9 **FDF v2.0 - Time Taken - RBMs - Variance.** The computational time to generate leads using RBMs remained largely predictable. The greater the number of features identified for initial investigation, the more time required to complete the calculations. Variance occurred due to the number of leads needing to be investigated. Note: HMM and ANN trials are not included. See Page 105. 112
5.10 **FDF v2.0 - Confidence - RBMs - Variance.** Confidence values generally display little variance. The learning rate of CDL operated as predicted in nearly all cases with one notable exception where the number of features produced by the fault injecting were smaller than previous trials. Investigation into the cause of this yielded inconclusive results. 112
5.11 **FDF v2.0 - Fault Position.** The average fault position when using the RBM appears to be competitive and often outperform other approaches. 113
5.12 **FDF v2.0 - Total Leads.** The total avenues for investigation are much higher when a lazy data ingest is used. 113
5.13 **FDF v2.0 - Fault Position - RBMs - Variance.** This graphs shows results of an identical feature being selected between trials. Normally, the first related feature to the root cause is selected from the list by a test administrator. In this case, only the original feature is selected to help illustrate variance. This helps provide an understanding of how the RBM makes suggestions. At 20 minutes all correct features are categorised within the top 10 leads. 114
5.14 **FDF v2.0 - Accuracy - RBMs - Variance.** The accuracy of the RBM approach increases whilst variance decreases with the exception of differences between 25 and 30 minutes. The reason for this is not fully confirmed. An initial investigation showed maintenance tasks being executed by the operating system at regular intervals set for every 30 minutes. This may create additional features for investigation and account for differences in the 30 minute trials. 114
5.15 **FDF v2.0 - Precision.** Precision of the RBMs show a marked improvement on previous ex-periments, however some ramp up time is necessary. Averages of results show a marked improvement between the 20 and 25 minute marks and a stronger trend than prior approaches. 115
5.16 **FDF v2.0 - Accuracy.** RBMs show an improvement in accuracy over other primitives. Initial values start out low, but higher than other approaches. They continue to show steady improve-ment with the possibility of meeting the accuracy of the HMM. 115
5.17 **FDF v2.0 - Precision - RBMs - Variance.** Precision tends to increase as more data samples are used. This coincides with previous observations at around 20 minutes where greater increases start to take hold. 116

5.18 **FDF v2.0 - F-Measure - RBMs - Variance.** A number of minor outliers occurred during the course of these experiments. Some tests provided a larger number of features to examine than were expected, and variance in fault index had an impact on results. 116
5.19 **Prediction Accuracy Formulas: UBL** 4.1: *True Positive Rate* 4.2: *False Positive Rate* 117
5.20 **Precision Measurements: UBL & the FDFs.** The precision of both FDF approaches remains low, however the RBM approach shows a promising trend as more data is added. UBL's precision drops the more data is added. The first three metrics show results for fault identification where all features above the correct root cause are considered false positives. The bottom two results (-ALT) show precision for fault detection. 120
5.21 **Average Position of Faults Based on Approach: UBL & the FDFs.** UBL and the FDFs prioritise potential sources of faults. Correct recommendations are represented as an average of all tests based on primitive type. Lower values signify better recommendations. 122
5.22 **Time Taken Performance Metrics: UBL & the FDFs.** The UBL experiment does not post timing data but instead reports performance as a function of total samples. Additionally, training times are also reported to last until each neuron has been updated 10 times making variance a possibility. This information is not given in the original study by Dean, et al [1]. The FDF experiments do provide timing data with greater variances being confirmed when switching from a greedy to a lazy ingest. . 123

List of Tables

2.1 **Autonomic Computing Levels, IBM, *circa* 2002.** This table represents the initial Autonomic Computing levels proposed by IBM, however a small addition of where Supervised, Semi-supervised, and Unsupervised learning methodologies has been appended. 22

4.1 **WMI Classes – Names & Unique Column Identifiers.** This table illustrates the classes and the columns used to uniquely identify rows within the sampled WMI data. 79
4.2 **Performance Tests – Names & Descriptions.** 81
4.3 **UBL Testing Suites.** UBL uses various testing suites on a number of application platforms. Although some of these tests are used across all scenarios, the majority are not. For simplification, a testing matrix is included here. 88
4.4 **Summary of Testing: UBL.** In the original work, results for UBL are presented textually, graphically, or sometimes not at all. *G* = Data via Graph Only, Blank = No Data, TP = True Positives, FP = False Positives. 95
4.5 **FDF and UBL Property Comparison.** This table presents a summary of the comparison of operating properties in unsupervised self-healing frameworks discussed in this chapter. 98

5.1 **Lead Times: UBL.** This chart represents the number of seconds UBL identified a failure before it reached a terminal threshold; higher values are better. Blank = No Data. 119

GLOSSARY

Adverse Configuration Change (ACC) A configuration change either simulated or real made by a human administrator that negatively impacts the performance of a computer system.

Artificial Neural Network (ANN) A family of statistical learning models used to estimate or approximate functions. Artificial neural networks are generally presented as systems of interconnected "neurons" which send messages to each other. The connections have numeric weights and biases that can be tuned using a learning algorithm and observations over time. This allows artificial neural networks to exhibit adaptive behaviours.

Application Programming Interface (API) A set of routines and protocols for interacting with software applications.

Case-based Reasoning (CBR) The process of solving newly observed problems based on those which are both similar and have been previously observed.

Contrastive Divergence Learning (CDL) A learning algorithm based on alternating Gibbs Sampling commonly used to train Restricted Boltzmann Machines.

Central Processing Unit (CPU) Computer circuitry used to carry out instructions of a computer program.

Direct Fault Injection (DFI) A type of fault either simulated or real that is not directly associated with intended use of a computing program or configuration change. Examples include unexpected thread termination, unexpected resource constraints such as out of memory exceptions, and other computing events typically outwith the intended use of software by a human administrator.

Domain Name Service (DNS) A service that resolves human-readable fully qualified domain names to internet addresses; *e.g.* www.google.com –> 173.194.45.48. DNS services underly much of the basic functionality of the internet.

Fault Detection Framework (FDF) A computing application used to detect anomalies within a computing system.

Genetic Algorithm (GA) An heuristic, iterative search algorithm that combines stochastic generation of new "populations" subsets with fitness tests to dynamically generate new, stronger outputs.

Generative Stochastic Network (GSN) A type of artificial neural network being pioneered using revisions to the backward propagation of errors learning algorithm.

Hidden Markov Model (HMM) A statistical model that attempts to forecast behaviours based on observed information – requires the system being observed to be a Markov process with unobserved (*i.e.* hidden) states – that is typically paired with the Baum-Welch learning algorithm.

Infrastructure as a Service (IaaS) A type of internet-based (*i.e.* cloud) service that provides access to hardware and basic, low-level software services. This allows for greater customisation potential for clients, but also more overhead. See "Software as a Service", and "Platform as a Service".

Integrated Development Environment (IDE) Any environment used for programming software on a computer system.

Internet Information Services (IIS) Microsoft branded service used for delivering web-based content (*i.e.* HTTP, HTTPS) on the internet.

Java Virtual Machine (JVM) A virtual machine used for running Java-based programming code. It is administrated by the Java Runtime Environment.

Java Runtime Environment (JRE) An environment for instantiating and controlling Java Virtual Machines.

κ-Nearest Neighbour (κ-NN) A supervised method for building a classification model using feature data. These models can be used to forecast computer feature behaviours.

Principal Component Analysis (PCA) An eigenvector-based multivariate analysis procedure that uses an orthogonal transformation to convert a set of observations of possibly correlated variables into a set of values of linearly uncorrelated variables called principal components. It is used primarily to reduce the dimensionality of data and determine principal components of interest.

Peer-to-Peer (P2P) An ad-hoc system of communication, typically between computers.

Platform as a Service (PaaS) A type of internet-based (*i.e.* cloud) service that provides access to a pre-determined platform for end users to interact with typically offering moderately developed, self-contained services. This allows for less customisation potential than IaaS offerings but also less overhead for use. See "Service as a Service" and "Infrastructure as a Service".

Quality of Service (QoS) A term to represent the overall performance of a computing service including availability, reliability, and similar measures.

Restricted Boltzmann Machine (RBM) A type of artificial neural network based on a Boltzmann Machine. It typically consists of hidden and observed layers of "neurons"; (see Artificial Neural Networks). Unlike fully recurrent neural networks, Restricted Boltzmann Machines do not allow direct cross communication between neurons on their respective layers.

Receiver Operating Characteristic (ROC) A graphical plot that illustrates the performance of a binary classifier system as its discrimination threshold is varied. The curve is created by plotting the true positive rate against the false positive rate at various threshold settings.

Service as a Service (SaaS) A type of internet-based (*i.e.* cloud) service that provides access to a service front-end and minimal infrastructure resources or platform customisation. This typically offers well developed, self-contained services with little to no customisation. (See Platform as a Service and Infrastructure as a Service).

Service Level Agreement (SLA) An agreed upon objective – usually between multiple entities – regarding computing performance.

Service Level Objective (SLO) A policy or other formal, written obligation regarding the intended minimum performance of a computing system or service.

Self-organising Map (SOM) A type of artificial neural network (ANN) that is trained using unsupervised learning to produce a low-dimensional representation of an input space. Self-organizing maps are different from other artificial neural networks in that they use a combination of neighborhood functions to preserve topological properties of the input.

Scalable Vector Graphic (SVG) A lossless image format that uses vectors to redraw scaled images upon demand.

Unified Modelling Language (UML) A General purpose modelling language used for describing the architecture of a software system and sometimes its associated processes.

Unsupervised Behavioural Learning (UBL) An approach for understanding errant computing system behaviours using unsupervised learning within a self-organising map.

Virtual Machine (VM) A simulated computing system typically running inside a hypervisor or larger (sometimes physical) apparatus.

Windows Management Interface (WMI) A proprietary service offered by Microsoft Windows for querying basic feature information.

Chapter One

Introduction

This thesis focuses on self-healing systems, how they operate, and advances in their respective detection strategies. It describes exigencies in this area of study and then demonstrates positive results from two separate experiments using stochastic primitives that leverage unsupervised learning. **Using a combination of unsupervised learning, stochastic primitives, and performance tests, the root cause of a fault in cloud infrastructure (*i.e.* virtual machines) can accurately be identified by comparing a system's observed and predicted feature behaviours.**

The increasing complexity of modern computing environments is continuing to produce challenges in reliable and efficient systems management. Complexity has convoluted administrative requirements such that the static capabilities of human-based supervision are showing decreases in their relative effectiveness [1, 2, 3, 4]. This is increasing the costs of systems management whilst simultaneously failing to address longstanding problems–such as issues with change management, and simple human error [5, 6]. These issues are particularly evident in multi-tier architectures where services comprise of several sets of systems with differing responsibilities.

The advent of self-adaptive systems is an approach in addressing the rising complexity requirements of systems management [7, 8]. These systems address multiple problems within this space from self-configuration (*i.e.* provisioning), self-optimisation, self-protection, and self-healing. Self-healing systems attempt to classify and analyse sensory data to automate the detection then mitigation of faults. This in turn reduces the need for systems to interface with human administrators, which presumably lowers operational costs and, ideally, improves upon existing mitigation techniques.

There are varying degrees of autonomy within self-healing systems. This is largely dependent upon the type of computing environment, management style, and learning algorithms or primitives used. The latter topic can broadly be summarised as the difference between reactive versus proactive (*i.e.* supervised and unsupervised) strategies, respectively. Reactive solutions are constrained to resolving faults only after they have been previously observed, *a fortiori*. In order to realise fully self-healing systems, a shift must occur from supervised to unsupervised learning strategies. Unsupervised strategies allow for this shift by anticipating faults in circumstances that have not been previously observed and, principally, offer the highest potential degree of reduction in human intervention. However, the use of such techniques come with costs – including potentially higher rates of error, and a lack of scrutability for some types of errors.

Additionally, criteria on the viability of self-healing – or self-adaptive – solutions have not been agreed upon. The simple fact of the matter is that there is no public information on what types of behaviours, error rates, or resource utilisations are acceptable for self-healing systems in production environments. One of the primary goals of this thesis is to encourage adoption of these techniques in such environments and demonstrate their potential viability.

1.1 Motivation

The realisation of self-healing systems presents a number of potential benefits in large-scale computing environments including the reduction of operational costs, faster fault mitigation than existing methodologies, and fewer problems related to complexity – such as human error in the consistency of systems' configurations and change control procedures [4]. However, there is presently no public data to support these claims, nor – prior to this work – are there any known studies that compare one type of self-healing system to another based on either performance metrics or their intended use. For those works that do exist within this area of research, their use depends on simulated data or they operate under artificial constraints.

Using stochastic primitives, performance tests, and unsupervised learning in conjunction allows for the accurate identification of problem areas within an infrastructure without having to train highly specialised members of staff – an action that is believed to be potentially cost reducing – whilst also shifting from reactive to proactive fault mitigation. Proactive mitigation promises faster resolutions than human counterparts, and reduction of down-time compared to reactive approaches.

Additionally, there are still no known public studies regarding human-subjects in this area.

However, it is anticipated that these types of studies will be more likely to occur once accuracy and timing expectations having been clearly set – which this thesis helps to establish.

This thesis provides the first known direct comparison of self-healing systems methodologies (*i.e.* frameworks) in order to establish a baseline, and it attempts to understand their respective specialisations in terms of computing environment, preference in primitives and learning algorithms, and their management styles. It then goes on to describe a novel approach for identifying the root cause of faults using stochastic primitives and unsupervised learning using non-simulated data in a fully reproducible manner. Results, code, and associated assets are provided to the public for the purposes of validation.

1.2 Hypothesis & Central Tenets

The general hypothesis and claims within this thesis are summarised here. Many of these statements rely upon further explanation or references that are provided later in their respective chapters.

The central hypothesis is listed here first, with central claims following:

1.2.1 Hypothesis

Using a combination of unsupervised learning, stochastic primitives, and performance tests, the root cause of a fault within virtual machines can accurately be identified in cloud infrastructure by comparing a system's observed and predicted feature behaviours.

Success is determined if an accurate list sorted by descending order of potential root causes is correctly returned upon instantiation of a fault with a known root cause in a controlled computing environment. A list will be accurate overall if it contains the correct root cause in the top 10 recommendations. Accuracy and precision will be measured by understanding the relative position of root causes in proximity toward the first position in the ordered list.

Failure is determined upon any result where the root cause of a fault is not immediately and accurately evident or it is below the top 10 – this includes any failure of the software in question, or its dependencies.

For constraint reasons at the minimum bound, the smallest dataset sample will be five minutes. This provides some amount of time for the system to build an expected behavioural profile.

1.2.2 Central Claims

1. Automated fault detection is important and has the potential to reduce operating costs and complexity in maintaining large-scale computing environments. This is based on the assumption that systems within such environments can detect and enact recovery strategies faster than their human counterparts, and that some environments – particularly auto-provisioning clouds – are becoming too complex to manage manually.

2. Using high-level performance tests is necessary to categorise observed data and it emulates and adheres to existing research within the field of administering systems using 'policies' rather than individual technical corrections [4].

3. This thesis assumes that: 1.) Some faults will not be detected by the framework application presented in this thesis, and 2.) by definition, it is unknown which faults will not be detected. For these reasons all of the generated outputs of the services described herein are evaluated manually to ensure accuracy.

4. Some faults are not recoverable – such as hardware errors. The types of faults anticipated in this book are limited to those that are recoverable only: adverse configuration changes (ACCs) (*i.e.* human error) and direct fault injections (DFIs) (*i.e.* intentionally crashing a service or application by corrupting its allocated memory).

5. Finally, this thesis is limited in scope. It does not attempt to make new research claims or changes to the state of the art regarding learning algorithms, stochastic primitives, or other, related areas of study. Although some premises are provided for the selection and use of the artificial intelligence techniques used in this thesis, operating theories behind learning algorithms, stochastic primitives, and other topics are not addressed in depth. Likewise, paradigm matters – such as if the primary focus of self-healing systems should be on fault tolerance or fault remediation – are intentionally not addressed.

   This thesis only focuses on providing a novel approach for the automated determination of the root cause of a fault under controlled conditions, and attempts to lay the foundation for further studies in this specific area of study.

1.3 Main Contributions

In summary, this thesis provides the following contributions:

1. A survey of self-healing systems that contrasts computing environments, learning algorithms, and management styles, (Chapters 2, 3)

2. A novel approach for self-healing systems that combines unsupervised learning with stochastic primitives to analyse feature changes in order to accurately generate a descending ordered list of fault hypotheses, (Chapter 4)

3. Foundational work for comparing fault identification approaches using unlabelled data, including the first known direct comparison of self-healing systems' approaches, (Chapter 4, Section 4.3 and Chapter 5) and

4. Results that show a demonstrable improvement in accuracy over existing approaches under similar conditions (Chapter 5).

1.4 Definitions

The majority of terms used within this book are primarily in reference to IBM's initial parlance, or those terms provided in Rodosek, et al's, *Self-Healing Systems: Foundations and Challenges* [9]. Definitions to terms based on these prior works may be occasionally updated to address current technological trends and research, with such instances being called out when appropriate. In some cases new terminology is provided to accent critical areas of research, or to call out important information – such as the difference between human error leading to faults (ACCs), and faults caused by unexpected program failures (DFIs).

Definitions of common terms are provided here for ease of reference:

Accuracy How close the measured value is to the actual (true) value. Depending on the context this is measured by whether or not the correct feature is in the top 10 features returned in the ordered list of leads, or by measuring all correct cases overall all sampled cases – $(N_{tp} + N_{tn})$ / $(N_{tp} + N_{tn} + N + _{fp} + N_{fn})$. Any features ordered higher in the list than one identified by the test administrator are considered false positives.

Availability Whether or not a computing system is accessible – *e.g.* via expected services.

Churn The rate at which computing device membership changes within a computing environment [10].

Clouds Collections of either real or virtualised computing devices that are centrally managed, and controlled by a single entity. Devices that exist as part of a cloud are more likely to

be configured identically, housed in a data-centre, and operated by a large professional or academic staff [10, 11].

Confidence A representation of a learning algorithm's perceived likelihood that a specific feature is related to the root cause of the fault – *i.e.* the probability of unexpected behaviour of a feature given some number of prior observations.

Note: Confidence value generation depends on the learning algorithm being used. For Naïve Bayes it is a weighted proportion of the previously viewed feature changes up to the maximum window size – *i.e.* 30. For Baum-Welch, an expectation-maximisation (EM) algorithm is used to find the maximum likelihood of potential behaviours using all of the observed set of feature inputs – this produces a proportional result with one value always equaling 100%. With Contrastive Divergence Learning (CDL), an "energy function" is used – see Section 2.3.3 and associated literature [12, 13, 14].

Drift A specialised term used to describe the phenomenon of the potential loss of accuracy between when a test completes as related to when feature data was last sampled from a computing system.

ElapsedTicks A programming object used for tracking time in C# that automatically accounts for differences in central processing unit (CPU) frequencies. This is not to be confused with "Elapsed Ticks", which are a similarly named programming object also used in C# for time calculations, but do not take into account differences in CPU cycles between computing systems.

ElapsedTicks are used to ensure greater reproducibility – minute differences in machine configurations and operating systems can impact the results gained from using traditional timing mechanisms. These differences occur regardless of hardware specification and can manifest even under identical configurations due to differences in timing frequency on system boot [15].

Fault Detection The process of determining when a fault is present.

Fault Identification The process of determining the root cause of a fault, if present.

Fault Position The position of the correct root cause of a fault in a list of fault hypotheses or "leads".

Feature(s) Any property – such as a performance metric or configuration value – in a computing system that details or describes a computing system's state. Examples include free disk space, an internet protocol address on a network interface card, a working directory, or the number of context switches being performed per second.

Grids A voluntary collections of either physical or virtual computing systems that share resources, and typically consist of multiple, heterogenous configurations. In these environments *churn* – the rate at which membership changes – is expected to be high, and systems are expected to be managed in an *ad hoc* fashion. This can translate to computing environments that do not require professional services to operate, such as those housed in a data-centre.

High Availability High availability is defined as having less than 5 minutes of service down-time in a normal 365 day calendar year or ~99.999% availability; also known as "*Five Nines*" in industry nomenclature.

Large-scale computing environment A distributed, possibly world-wide, collection of computers that consists of many different layers of components. This term is kept intentionally ambiguous to address a plethora of acceptable solutions and conditions and avoid over specialisation.

Although there are numerous interpretations as to what constitutes a computing environment the majority of terms applied within this thesis are taken from a single source [11].

Performance Test An human supplied set of logical conditions capable of being discretely evaluated which indicate whether or not a computing system is operating within expected and acceptable boundaries (see Service Level Objective(s)). Sometimes discussed in association with *Fitness Tests* when discussing Genetic Algorithms.

Precision How close the measured values are to each other.

Root-cause Analysis The act of determining the source or sources of a particular fault; in this case the determination of which feature or set of features triggered or contributed to said fault such that a service-level objective or performance test failed to be met.

Reliability A percentage of time that a computing system operates as expected.

Self-healing Systems Self-healing systems are defined as servers – either physical or virtual – that detect and recover from faults without in an automated fashion without interrupting the overall usable state of a service where possible; some faults are assumed to be unavoidably service impacting.

Specifically, self-healing systems are virtual machines that are intended to exist within a large-scale computing environment under the above definition. These environments are assumed to host a service requiring high-availability.

Stability How fast a computing system can mitigate faults and return to its original state.

Standard Computing Environments (*i.e.* Traditional computing environments) are defined as established, non-virtualised and typically legacy computing environments that are otherwise similar in definition to *clouds*.

Stochastic Primitive A special, collective term for computing primitives that leverage a random probability distribution or statistical model to forecast and approximate outputs of a given function. This typically includes Artificial Neural Networks, Hidden Markov Models, and Restricted Boltzmann Machines.

Supervised Learning The use of labelled data to train a computing primitive to infer a function. Typically, this is achieved by pairing an expected output with an input.

Time-Taken The total time required in *ElapsedTicks* between service-level objective (SLO) failure and print out of an ordered list of fault hypotheses to an output screen or terminal (see *ElapsedTicks*).

Total Leads The total number of fault hypotheses generated by a stochastic primitive at the time of SLO failure. This represents the total number of avenues to be explored by the Fault Detection Framework (FDF).

Unsupervised Learning The use of unlabelled data to train a computing primitive to infer a function. Data is provided to the primitive without reinforcement or an expected output. This allows the primitive to build representations of the input for later decision making [16].

1.5 Published Works

This thesis incorporates work that has been previously peer-reviewed, published, and presented:

1. C. Schneider. "Autonomic Techniques for Systems Management" (Poster Session). *Sixth International Workshop on Self-Organizing Systems* (IWSOS) 2012. Delft University of Technology. Delft, The Netherlands. 15-16 March 2012. [17].

   This work was primarily focused on fault discovery mechanisms in self-healing systems, and, broadly, discussed what autonomic techniques were currently in place for their subsequent self-management. An overview, motivation, and summary of the problem space as it existed in 2012 were provided along with a preliminary description of the approach to be taken in subsequent research (Chapters 4, 5).

2. C. Schneider, A. Barker, and S. Dobson, "A survey of self-healing systems frameworks", in *Software Practice and Experience*, Wiley, 2013. [7]

   The journal of *Software Practice & Experience* published the literature survey included in this book (Chapter 2). It provides an overview of the history of the Autonomic Computing initiative before focusing specifically on self-healing systems and a contextual examination of their implementations.

3. C. Schneider, A. Barker, and S. Dobson, "Autonomous Fault Detection in Self-healing Systems: Comparing Hidden Markov Models and Artificial Neural Networks", in *Proceedings of International Workshop on Adaptive Self-tuning Computing Systems*, ADAPT '14, (New York, NY, USA), pp. 24:24–24:31, ACM, 2014. [18]

   This is the first publication presenting the approach described in this thesis. It incorporates the design, development, and metrics gathering processes associated with subsequent experiments, and provides a baseline for the more complex approach utilised in the following paper.

4. C. Schneider, A. Barker, and S. Dobson, "Autonomous Fault Detection in Self-healing Systems using Restricted Boltzmann Machines", in *11th IEEE International Conference and Workshops on the Engineering of Autonomic Autonomous Systems*, (Laurel, Maryland), IEEE Computer Society, IEEE, 2014. [19]

   This paper represents the state of the art as proposed in this thesis for generating fault hypotheses using stochastic primitives and unsupervised learning in self-healing systems. It compares results with the previously mentioned approach before laying the foundation for future work.

5. C. Schneider, A. Barker, and S. Dobson, "Evaluating unsupervised fault detection in self-healing systems using stochastic primitives," *EAI Endorsed Transactions on Self-Adaptive Systems*, vol. 15, January 2015. DOI:10.4108/sas1.1.e3. [20]

   This is a summary paper published in the journal *EAI Endorsed Transactions on Self-Adaptive Systems* that discusses and contrasts both prior works with an external experiment that has similar goals. It contributes and provides an approach for practical implementation and compares performance metrics against a related study in self-healing systems research.

1.6 Organisation

This thesis is divided into five chapters and one appendix. Chapter 1 introduces self-healing systems, the motivation and claims discussed herein, and provides an overview of contributions and structure within this thesis. Chapter 2 summarises prior research by contrasting the management styles, computing environments, and learning algorithms of existing self-healing systems before concluding with exigencies in the field, an in-depth motivation, and an hypothesis. Chapters 3 and 4 describe the approaches used to evaluate the hypothesis, including theoretical assumptions, technical specifications, and implementation details of the experiments, and discusses their results, respectively, in chronological order of publication starting with Artificial Neural Networks (ANNs) and Hidden Markov Models (HMMs), then by Restricted Boltzmann Machines (RBMs). The final chapter concludes with findings, lessons learnt, and future avenues of research.

Chapter Two

Background

This chapter discusses the history of self-managing systems – chiefly from the context of IBM's Autonomic Computing Initiative. The following chapter builds on this information to expand the motivation provided in chapter one.

2.1 Introduction

Self-healing methodologies are often realised through the use of machine learning techniques or other aspects in artificial intelligence. They have have been described via architectural differences [21], network behaviours [8], research areas [22], biological likenesses [23], and, most recently, by contrasting their learning algorithms, implementation, and management styles [7]. These surveys have produced a broad spectrum of knowledge and highlighted notable advances and exigencies within the field. However, the effectiveness of these solutions and the commonalities shared between implementations have not yet been fully explored.

This chapter discusses the background of such systems, and helps to lay the foundation for further exploration in comparing self-healing systems methodologies. It uniquely divides self-healing behaviours contextually which is based on the idea that not all approaches are created equal, nor appropriate given the purposes of their implementations.

The type of environment or infrastructure in which self-healing frameworks operate, the self-healing behaviours or problems expected to be addressed, and their manageability requirements or hierarchical needs are critical for understanding self-healing systems research and methodologies. These factors are categorised herein as computing environments, learning

methodologies, and management styles, respectively. Analysing self-healing frameworks based on commonly shared use-cases (*i.e.* tiers) allows for a comparative understanding of each methodology, their respective benefits, and their relative human costs. By contrasting behavioural properties with their expected implementation and level of autonomy, this chapter provides a greater understanding of which techniques are being leveraged, and under what circumstances. It also examines correlations between these factors by exploring the type of self-healing methodologies as related to their expected environment.

2.1.1 Terminology

Although some definitions have been attempted, the terminology used within self-healing systems is not fully agreed upon [4, 9]. This has caused confusion when similar or sometimes identical terms are used under different connotative assumptions. This is particularly evident in self-healing systems where common goals are shared but approached under different ideologies-such as self-*, self-managing, mimetic, and evolutionary computing. When this happens definitions can have unexpected cross referencing problems – such as those between self-healing, and self-configuring.

Ambiguity in terminology is also a major issue. Self-managing systems–a term broadly associated with systems that can monitor and adjust their own behaviours–represents a large area of study. As such, some approaches divide self-managing systems into tenets–*e.g.* IBM's *Self-** approach [2, 4]. IBM's tenets are categorised into *Self-healing*, *Self-configuring*, *Self-protecting*, and *Self-optimizing* behaviours. Self-healing systems, specifically, are defined as

> *"... systems [that] discover, diagnose, and react to disruptions. For a system to be self-healing, it must be able to recover from a failed component by first detecting and isolating the failed component, taking it off line, fixing or isolating the failed component, and reintroducing the fixed or replacement component into service without any apparent application disruption.*" [4] [p. 8].

However, what constitutes the successful implementation of self-healing systems is much more equivocal.

In 2003 Ganek and Corbi state "*...self-healing and self-configuring is the ability to dynamically insert new pieces of software and remove other pieces of code, without shutting down the running system.*" [4] [p. 14]. This definition highlights the ambiguity between the differences in these tenets, but it also highlights several problems with the definition provided by Ganek

and Corbi: It does not readily address the current trends in technology, that some faults are unpreventable and will require a system shutdown to mitigate, or that some faults may not be predictable (but still recoverable).

Technological trends have shaped the way in which self-healing systems are being defined. The rise of mass virtualisation post 2003 has arguably allowed for large-scale computing environments to accept the arbitrary shutdown of one system to be replaced by a new, better configured virtual instance without interruption to live, production services. It is this environment upon which the theories in this book are largely based. However, these theories are built with the expectation that if it works in virtual environments, it should also work in physical environments.

Some faults will not be preventable – such as hardware failures, software corruption, or poor decisional choices by human agents. In these circumstances a system shutdown may be unavoidable. Such circumstances should not invalidate the legitimacy of self-healing systems as an approach.

Some faults are not predictable but are still recoverable. In those instances the success of a self-healing system should be defined by evaluating its enactment of a self elected course of action– or *recovery strategy*. A recovery strategy should be at least equivalent to a human counterpart in terms of mitigating the faults, and ideally implemented in a faster, and yet still accurate fashion.

Under these considerations the latter stipulation – that a system cannot be shut down to address a fault – is considered to be obsolete. Additionally, because Ganek and Corbi's definitions do not address current technological trends, they do not feel well enough defined to be of sufficient use.

If a self-healing system automatically corrects a fault by changing a program's feature behaviour, is that a form of self-configuration? What if that feature behaviour is adjusted through a configuration file? The vast majority of self-healing systems experiments leverage autonomous reconfiguration as a mechanism for addressing faults – either before or after they happen. Broadly summarised, key approaches consist of local parameter tweaking (including evolutionary, bio-inspired, and search-based techniques) [18, 19, 24, 10, 25, 26, 27, 28, 29], behavioural correlation [30, 31, 32], contextual weighting of information (*i.e.* windowing) [18, 19, 33, 34, 29, 35], self-election of roles based on availability and load [36, 37, 38], and atomistic reconfiguration – the independent discovery and use of openly exposed resources [39]. Notably, a number of frameworks implement more than one approach – particularly parameter tweaking which is nearly universal.

In fact, the changing of a system's overall state in any self-elected manner – which happens almost continuously in most systems anyway – provides ambiguity to the term in general. Self-healing and self-configuring in these respects are, arguably, almost synonymous in definitions in such cases.

Self-configuration does remain a unique and valid subset of self-management – however, due to confusion between the defined areas of focus of the aforementioned tenets [4], it behooves us to expand on and clarify the initial definition of self-healing systems. Initially, self-configuring systems were described in the following manner:

> *"When hardware and software systems have the ability to define themselves themselves 'on-the fly,' they are self-configuring. This aspect of self-managing means that new features, software, and servers can be dynamically added to the enterprise infrastructure with no disruption of services.*" [4] [p. 8].

In modern parlance, this behaviour is more akin to *self-provisioning* – the autonomous instantiation and adoption of a configuration subset (or role) within an infrastructure. Many cloud computing environments accomplish this behaviour through *provisioning managers* – but their specifics are varied and are outside of the scope of this work.

Self-healing systems are then taken to mean any system leveraging a framework that autonomously detects and then subsequently generates a recovery strategy from said fault – where possible. Caveats to this definition include the ability to detect faults that are not able to be mitigated, and that some faults cannot (or occasionally will not) be detected before they occur. That is to say a system need not detect every potential fault perfectly or risk not being able to be defined as a self-healing system.

The decision to define self-healing systems in this manner is not arbitrary and is based on prior work as described in the following Section 2.1.2, and in Section 1.4.

2.1.2 Assumptions

The definition of self-healing systems has been expanded to include behavioural aspects that are commonly evaluated in modern computing infrastructures. It is no longer acceptable for a system to simply detect and recover from faults – it must do so transparently, and within certain performance criteria. As such, some assumptions about how self-healing computing systems should operate have changed since 2001.

The integration of behavioural aspects has helped to unify business needs with IBM's original vision of self-healing systems. By adopting partially self-healing systems into traditional infrastructures, an evolution of techniques and new self-healing systems methodologies have emerged. However, not all self-healing methodologies are compatible with existing infrastructures and the maturity of many of these techniques has not been fully realised. As self-healing systems methodologies become more mature, less human supervision should be required.

One approach to understanding maturity in a self-healing environment is by evaluating systems state *via* behavioural properties [9, 40, 6]. By understanding when and how long a system executes self-healing behaviours, it becomes possible to evaluate self-healing approaches against existing implementations. Understanding the effectiveness of self-healing computing systems against current approaches provides a practical baseline for understanding the advancement of self-healing systems outside of the *Autonomic Maturity Model*.

Although there are numerous physical components that make up large-scale computing environments, the scope of this thesis primarily emphasises virtual servers as central points of focus. It is important to note that exigencies can exist outside of this scope, which the server is still responsible for identifying. Examples of this include network connectivity diagnosis, and being able to determine resource availability, such as a remote application programming interface (API).

Devices in these environments are expected to have high-availability constraints, and be relatively static in terms of their rate of churn. Typically, large-scale computing environments utilise multi-tiered architectures divided into front-end, middleware, and back-end sub-divisions that exist absent of virtualised components. Standard (*i.e.* traditional) computing environments are intended to represent the most common configurations for small, mid, sometimes large-size network-aware service applications.

It is assumed that self-healing behaviours in computing environments may never be fully realised and that some problems will indefinitely require human interaction. Although this is not in keeping with the initial proposal, at some point it is perhaps unavoidable. For example, there are no known software solutions to mitigate non-redundant hardware failures. However, diagnosing and escalating such a situation to an administrator is still a desirable self-healing behaviour. As such, systems that can operate to the edge of their limitations are still considered to be successfully self-healing.

That being said, shifting from supervised to unsupervised learning is assumed to be more likely to produce fully automated self-healing behaviours. Supervised approaches, by definition, can only respond to situations retrospectively and are not the most efficient mechanism for reducing

costs as they still require human interaction [28].

As *large-scale* computing infrastructures have become more complex, existing methods for operating and maintaining systems have become less effective [4]. Anecdotal evidence suggests that the use of skilled engineers to apply monitoring techniques that search for faults, engage in root-cause analysis, and execute appropriate recovery strategies remains the *de facto* standard of most professional organisations. Most of these monitoring techniques utilise some form of behavioural test to indicate when a fault is present. Self-healing systems seek to automate these processes. If a service fails, rather than requiring an engineer to intervene, a self-healing system would autonomously diagnose the fault and then execute a recovery strategy.

Lastly, recovery is assumed to be a more difficult problem than detection – as Kephart said: "*The final stage, automated re-mediation of a problem once it has been localized, is perhaps the most difficult*." [5]. However, the detection of faulty states is necessary before executing recovery strategies, *a fortiori*. This logic is the foundation upon which some aspects of framework maturity are gauged – a topic discussed further in Section 2.2.1.

2.2 The History of Autonomic Computing

Many of the methodologies discussed in this paper refer to existing works in Autonomic Computing. Autonomic Computing covers a wide range of topics in self-managing systems– including *self-healing*, *self-optimisation*, *self-protection*, and *self-configuration* properties. Although a familiarity with this area of research is assumed, a summary of foundational literature is provided here for ease of reference.

This section discusses in brief the Autonomic Computing Initiative [2], and the goals and criteria of self-healing systems, as initially described by IBM and subsequent publications [4, 3]. The illustration of these goals provides a way to narrow the problem space into addressable components and brings context to the methodologies presented in this survey.

The Autonomic Computing Initiative was proposed in 2001 to address growing complexity in systems management [2]. IBM proposed building software that could autonomously manage systems using a series of closed control loops and *environmental knowledge* per the work of Dave Clark, et alia [41]. *Environmental knowledge* is often denoted simply as *K*. Recursive software elements combine contextual information (*i.e. K)* with a series of inferential steps to make real-time decisions that mitigate problems and automate palliative maintenance tasks. Over the last 10 years several advances have been made in realising these goals.

2.2.1 Self-* Systems

In 2003, IBM published two articles that built upon their initial proposal outlining the aforementioned four primary tenets in Autonomic Computing, a general process for autonomic systems management [3], and a set of criteria that described behavioural levels and generic goals of self-managing systems [4]. The process for automating systems management tasks, often referred to as MAPE+K, outlined a recursive approach for continuously understanding and making changes to a system's state. By utilising *knowledge* (*K*) about a system's environment, a designated software agent would: Monitor, Analyse, Plan, and Execute (MAPE) instructions to meet user-specified policies. Since its introduction, MAPE+*K* has proven to be a central component in many self-managing systems implementations.

In order to understand the effectiveness of a given MAPE+*K* based process, behavioural benchmarks (*i.e.* levels) were used to evaluate the implementations' maturity [4]. These levels ranged from basic to fully autonomic and were evaluated based on whether they could consolidate information, recommend an action, autonomously take an action, and finally interpret a user-specified policy to do all of the aforementioned behaviours (Figure 2.1). Importantly, this article recommended an evolutionary approach in reaching each of these stages. Building self-managing systems that operate at different levels permits heterogeneous infrastructures, and allows for the gradual adoption of Autonomic Computing technology. This includes environments where existing systems may not be compatible with all of the autonomic computing levels.

To address the challenges proposed in these two articles, agent-based approaches for managing systems were introduced [6]. Utilising aspects in artificial intelligence, this work was based on an earlier text discussing reflex, goal, and utility agents [42]. Simply stated, reflex agents use if-then rules to map actions to a specified state. In practice, this approach is used once some condition is met to execute a pre-specified set of instructions. Goal and utility-based agents attempt to exhibit rational decision making by autonomously determining what actions to take based on expected results. The primary difference between goal and utility based agents is that the former selects behaviours to attain a given objective, whilst the latter attempts to reach and optimise behaviours such that suitable trade-offs between these multiple objectives can be achieved at once. This was particularly useful if two goal policies contradicted each other.

Using this approach as a foundation, IBM proposed that self-managing solutions leverage *Action*, *Goal*, and *Utility* policies. These policies incorporated high-level objectives with systems tasks whilst allowing for resolution conflicts between enacted behaviours. However, the implementation of broad level policies have produced challenges in evaluating the effectiveness

Autonomic 'Level'	Description
Basic	► Multiple sources of systems generated data. Requires extensive, highly skilled IT staff.
Managed	► Consolidation of data through management tools. IT staff analyses and takes actions – *Supervised* methodologies.
Predictive	► System monitors, correlates, and recommends actions. IT staff approves and initiates actions – *Semi-supervised* methodologies.
Adaptive	► System monitors, correlations, and takes action. IT staff manages performance against SLAs – *Unsupervised* methodologies.
Autonomic	► Integrated components dynamically managed by business rules/policies. IT staff focuses on enabling business needs.

Table 2.1: Autonomic Computing Levels, IBM, *circa* 2002. This table represents the initial Autonomic Computing levels proposed by IBM, however a small addition of where Supervised, Semi-supervised, and Unsupervised learning methodologies has been appended.

of self-managing systems. In the following year, a framework called DTAC was introduced for evaluating the performance of a self-healing system [43].

DTAC unified the MAPE+*K* control loop with industry requirements, and provided a baseline for performance metrics for evaluating self-managing systems. It described and quantified properties such as stability, accuracy, settling times, and efficiency. By using these properties it became possible to conduct behavioural evaluations based on a system's environmental knowledge, and historical performance data. The evaluation of this information led to a more expansive approach that discussed general research challenges in self-managing systems, and a variety of scientific advances in self-managing systems [5].

Specifically, self-managing systems solutions were divided into elements, systems, and interfaces, and standards definitions and requirements for each of these components were proposed. This helped to unify the mission of Autonomic Computing with practical implementations by illustrating examples of where action [44, 45], goal [46], and utility policy approaches had been implemented [47, 48, 49]. Technologies related to these policies varied from symptom matching [44] and task scheduling [46], to more complex approaches such as event correlation with performance metrics [47, 49].

Notably, Kephart argued that the division of self-managing systems into autonomic elements would allow for easier adoption of legacy systems. By incorporating existing services with an autonomic interface, legacy architectures could be made to adopt self-managing strategies.

Once a legacy system had an access point for autonomic communications, self-managing systems could exert some influence over the existing infrastructure. Indeed the notion of inter-element communication was arguably the central thesis of this paper:

> "*The main new research challenge introduced by the autonomic computing initiative is to achieve effective inter-operation among autonomic elements. In order for this to happen, product developers must look beyond their natural product-centric tendencies and cultivate a more holistic, system-level point of view. In other words, specific autonomic elements must be designed with a greater awareness of the fact that they will be situated in autonomic systems and intercommunicating and interacting cooperatively with other autonomic elements.*" [5][p. 2]

The challenge of reliable inter-operation and systems communication continues to be an open problem in self-managing systems, under which self-healing systems research is frequently categorised.

The establishment of core tenets, the MAPE+K process, evaluation methodologies, the autonomic maturity model, and action, goal, and utility policies, created a foundation for further contributions in self-managing systems. The ideas have also migrated into the domain of communications [8], and the progress made in the last 14 years has been largely summarised [1]. However, as the field has matured and new technological advances have been made – such as systems, and environment virtualisation, and the rise of mobile platforms – research in self-healing systems has diverged and become more specialised. There are now different types of self-managing systems based on these contexts.

2.2.2 Self-Healing Systems

To achieve the goals of self-healing systems a set of criteria must first be defined that is present in a majority of self-healing systems methodologies that are to be evaluated both now and in the future. It is for this reason that computing environment, learning methodology, and management style were selected for comparison. Each of these properties exists in the prior literature in some form making them easier to classify. Additionally, the effectiveness, capabilities, and contextual uses for those systems are more readily captured.

It is no longer sufficient that self-healing systems can be analysed without first analysing and understanding their intended purposes and requirements. A comparison of their effectiveness

and categorisation of their uses must first be provided to understand the progress self-healing systems have made, and to establish a baseline for future analysis.

2.3 Machine Learning Techniques

A number of machine learning techniques are used in the completion of the included experiments. Chiefly, the use of statistical models that contain adaptive weights and biases turning by learning algorithms make up the technical bulk of the discussed approaches. This section briefly covers the structure and update mechanisms of the three stochastic primitives used in this thesis to provide the reader with a general understanding of their operation. However, many of these approaches are dependent upon prior works. Details of how the primitives operate beyond a basic level, including mathematical background, history, and theory, are left to the reader to explore at their discretion.

Stochastic primitives consist of three major components: a method for adjusting weights, a *model*, and some mechanism for approximating non-linear functions of a given input. Weighting mechanisms can consist of multiple components such as a learning algorithm and an activation function. Their primary purpose is to take input and reinforce paths along the model in a consistent and predictable manner after some event has been observed. This can occur using both closed and directed cycles, but for complexity and resource constraint reasons solutions presented in this thesis are limited to the latter. The collective method for adjusting weights and their associated models are sometimes called a *module*.

In call cases within this thesis a stochastic primitive consists of a *Visible* layer that represents the actual behaviour of the computing system, and a *Hidden* layer – a computed, abstract representation of the observed data. By correlating observed behaviours and mapped cases, a stochastic primitive can leverage a learning algorithm to reinforce its paths or probabilistically forecast information. Understanding how accurate and efficient these learning algorithms are, within the context of self-healing systems, one of the primary goals of this study – *i.e.* Did the learning algorithm when used under this specific *model* successfully correlate the root cause of a fault based on feature behaviours?

Models represent the structure of the primitive. They define what paths are open for communication between a stochastic primitive's different layers, and contain the weight values and pathing objects (*i.e.* "neurons"). For operational reasons, learning algorithms are associated with a specific model. This is because some assumptions must be made from which the learning algorithms can infer information and where that information is to be stored – in this case, in a

float that represents a weight on a neuron.

Function approximation can occur through numerous different methods, typically via target functions. Target functions attempt to match the output of an observed state and then derive the input. Differences in these functions vary substantially between primitives and are the bulk of the following discussion.

2.3.1 Artificial Neural Networks

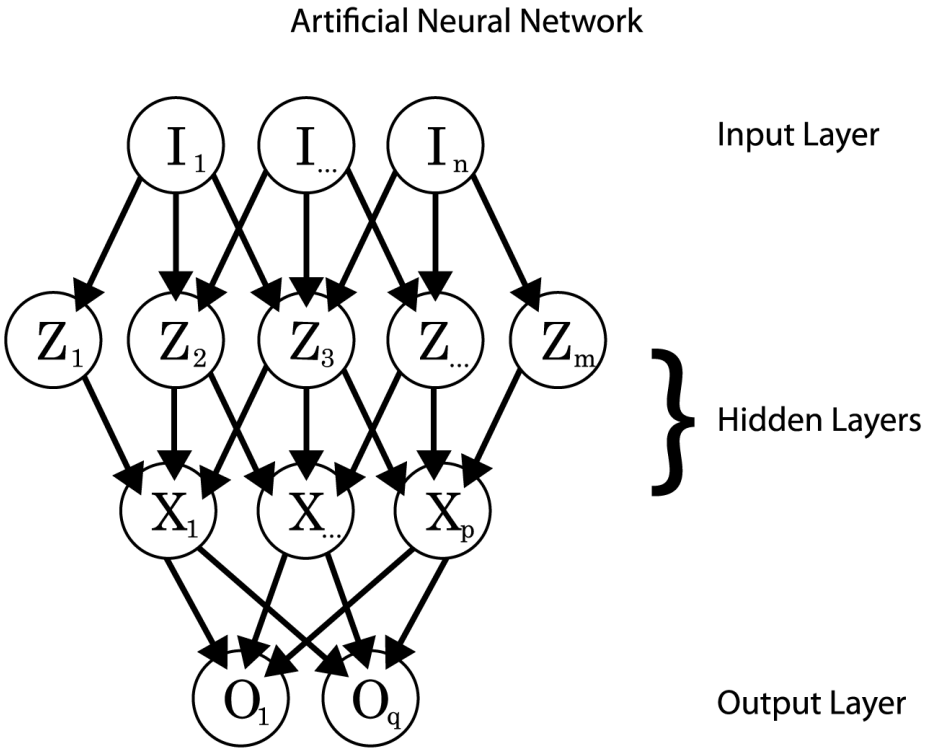

Figure 2.1: Artificial Neural Network. ANNs are a type of statistical model that operates by updating weights along paths between hidden and visible layers to forecast or otherwise 'learn' a series of inputs.

A number of different computing primitives can be categorised as ANNs, but for the purposes of this thesis they most closely resemble a multi-layer network consisting of Sigmoid neurons. A Sigmoid neuron uses a non-linear transfer function to determine activations for evaluation against a step function that uses a weighted, moving average. In this case, u represents in all cases to the weight sum of n inputs to the neuron, where w is a vector of *synaptic weights*.

$$u = \sum_{i=1}^{n} w_i x_i$$

A step function then evaluates whether or not the neuron activates (*i.e.* along a given path). If the given sum is above some threshold (θ), then the neuron activates thus changing its eventual output.

$$y = \begin{cases} 1 & : u \geq \theta \\ 0 & : u < \theta \end{cases}$$

A number of approaches exist for updating the threshold value. Which approach is used depends in the type of ANN that has been implemented.

For example, in this case θ is hard-coded at 0.80; a value determined by a $\frac{4}{5}$ success rate in the minimum sample set size for the experiments. This is done such that ANNs in this thesis are not able to operate until they have at least 5 samples to predict from, and that a potential root cause will not be selected unless at least this many observations has been met. It also provides for an arbitrary measure to truncate potential outliers.

Weight updates to the neuron occur in this case through Naïve Bayes. In the simplest of terms, this means the previously observed state is assumed to be the most likely observed state in the future adjusted proportionally by the number of observed states that match the prior observation, over the number of total observations. In this case, the total number of observation is limited to the 30 – a topic discussed further in Section 4.6.

2.3.2 Hidden Markov Models

Like ANNs, HMMs divide observed and unobserved (latent) information into *Visible* and *Hidden* layers, respectively (Figure 2.2). HMMs operate by forecasting the hidden layer (Zs) using observations from a visible set of inputs (*i.e.* a Markov chain, Xs).

There are numerous training algorithms that can be used with HMMs, such as the Vertibi algorithm. With Vertibi, the goal is to find the most probable sequence of hidden states given a set of visible states. This allows for multi-step ahead forecasting given a some number of previously observed behaviours. A simplified version of this algorithm exists called Baum-Welch, which allows for single step-ahead forecasting. Although both are algorithms are discussed, it is Baum-Welch which is used the experiments contained within this thesis.

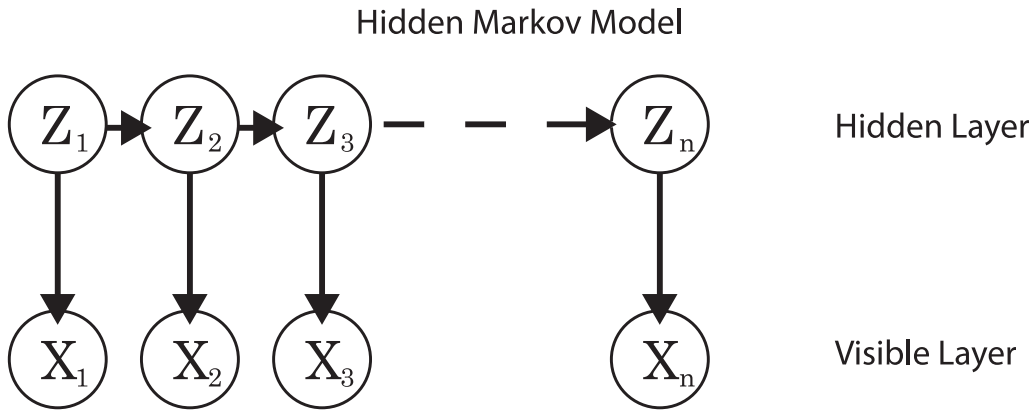

Figure 2.2: Hidden Markov Model. HMMs operate by forecasting the hidden layer (*Z*s) using observations from a visible set of inputs (*i.e.* a Markov chain, *X*s). Unlike ANNs, HMMs do not use more than two independent layers to separate observed and hidden data.

Baum-Welch operates by using the joint probability of a collection of *Hidden* and observed discrete random variables to find the maximum likelihood of an observed state. It assumes that the hidden variable is independent of previous hidden variables and that the current observation variables are only the result of current hidden state. By using expectation–maximisation (a topic not covered in this thesis), forecasts can be made using historical observations.

In short, given a number of observations, Baum-Welch attempts to final a local maximum for θ, such that the probability of the given observed states is satisfied provided some previously observed parameter. This local maximum is determined via the recursive forward-backward propagation of errors, that describe, respectively, the probability of observing some event at a specific time, and the probability of a given sequence as compared to a related observed series.

Afterwards, weights are updated based on probabilities as described in Bayes' theorem [50].

2.3.3 Restricted Boltzmann Machines

A special type of ANN called a RBM takes the two aforementioned approaches a bit further by using a form of *Alternating Gibbs Sampling* [51] called CDL [13]. Rather than using a number of subsequent layers for training RBMs organise themselves into a two dimensional graph; one top and one bottom layer.

The top layer acts as the hidden layer seen in previous approaches and corresponds to the interpreted model of observed behaviours. The bottom layer is directly linked to observed behaviours and serves as a guiding point for training data, without the need for supervision. Or,

intuitively explained, on updating the top layer the RBM attempts to model observed behaviour, and on updating the latter, the 'real' data is updated to correct any errors – the goal is to match the top layer to the bottom layer in terms of output as quickly and cheaply as possible.

RBMs use an energy-based model as opposed to a maximum-likelihood approach. Energy-based models associate a scalar to each observed state a variable of interest. Learning corresponds to modifying that activation potential such that its shape has desirable properties similar to the Sigmoid function typically ascribed to ANNs.

The adjustments made to these energy-based functions are outside the scope of this thesis, but they form the basis for adjusting weights and biases. In summary, RBMs can be thought of as log-linear Markov random fields for which the energy function is linear in its free parameters. To make this more powerful, an assumption is made to move from linear to non-linear by assuming that some properties are never observed (*i.e.* hidden), and we disallow direct communication or updates between neurons on the same layer.

Energy functions for RBMs are defined as $E(v,h)$ where $E(v,h) = -b'v - c'h - h'Wv$. W in this case represents the weights of the connecting hidden and visible neurons, and b,c are the offsets – gradual, co-related increases and decreases – of the visible and hidden layers' values, respectively.

The structure of an RBM provides it an advantage over HMMs – conditional independence of the visible and hidden neurons (figure 2.3). This allows for a cheap, multiplicative update mechanism for said variables above which are typically much more expensive in other primitives. When using binary inputs, such as those used in this thesis, it becomes very cheap to update and adjust weights through the associated activation (energy) function. Further details on this can be found in a number of resources [13, 12].

As a high-level overview on RBM's weight update process, it first starts by using a training vector on the visible layer's neurons. It then alternates between updating all the hidden units' weights and biases in parallel via the use of said energy formula and Monte Carlo sampling before doing the same thing with the visible units. Details on this process are long but can be found in external resources [52]. The updating process occurs until a specific number of time cycles (*i.e.* epochs) are spent.

CDL allows the RBM to train in near real-time, and to do so more efficiently than previously discussed primitives. Because CDL is good at using generative weights to convert posterior distributions to learn, it can tier itself such that it never has to learn to model the posterior distribution over the hidden units. In the simplest of terms, most learning algorithms aim to

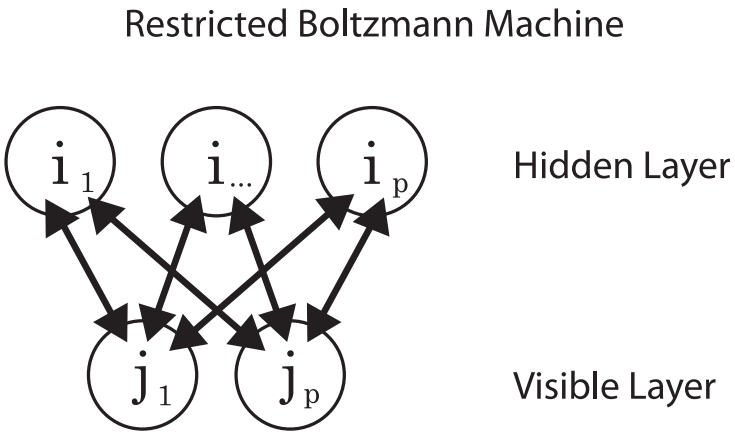

Figure 2.3: Restricted Boltzmann Machine. RBMs operate similarly to ANNs and HMMs, but adjust their weights by using an form of alternating Gibbs sampling. This allows them to update their layers in parallel – an advantage over other stochastic primitives.

be good at the former, but are bad at the latter. The RBMs can avoid the major difficulties in modeling the posterior distribution over the hidden units.

Additionally, RBMs operate similarly to HMMs by using Markov chains as inputs. However, they have a distinct advantage in that adding additional layers provably improves the model of the input data – although this increases their usage costs. The proof their continual improvement is complex and omitted from this work but is described via parallels to "variational free energy" via Hinton, *et alia* [14].

2.3.4 Stochastic Primitives

Performance for stochastic primitives is primarily measured in terms accuracy – including correct behavioural inference given an input, and the exclusion of extraneous information. It is also sometimes measured via the ability to synthesise a series of inputs given an output – as is the case with the Vertibi and CDL algorithms. To achieve this in all settings – including noisy scenarios – is challenging and no single approach has proven completely reliable.

As previously stated, stochastic primitives are a type of statistical model governed by a series of rules and functions. They can use both closed and directed cycles to adjust weights along paths formed by these functions to form maps. The maps represent relationships between observed and inferred information – in this case, changes in a feature's behaviour and the state of a system's health.

Maps can consist of paths, or be literal geometric maps in the case of self-organising maps (SOMs). Different primitives are more suitable for different problems depending on their expected outputs. *e.g.* If the goal is to find the smallest overlap shape possible using a connected graph, SOMs will probably be a better solution than HMMs.

Stochastic primitives are the objects of choice for this thesis because of their malleability and their established research record in forecasting behaviours. Specifically, they can be readily adapted to a number of different circumstances and their use is well established with autonomously analysing and classifying data.

An alternative to this approach would have been to use Support Vector Machines (SVMs) or Linear Classifiers (LCs). Although wide-spread and historically popular amongst researchers for solving similar problems, these primitives primarily use supervised learning which makes them unable to exhibit the highest degree of autonomous behaviour in self-managing systems. The lack of suitable alternatives at the start of this research was one of the driving forces in this area – although it is proving to be a more popular approach. This has been seen through the adoption of this approach in existing self-healing systems environments, such as WS-* [53], and for generic problem solving using performance based metrics [54]. One experiment even seems to replicate many of the results in this study [55].

The experiments in this thesis could have been conducted sooner but research into some aspects of stochastic primitives has been deterred due to technical challenges. The advent of the XOR circuit and the costly mechanisms involved with its implementation in the 1960s and 70s, discouraged the initial use of ANNs due to complexity constraints [56]. Resource utilisation also represented another issue which is, comparatively, large. Although these costs have not been lowered, available resources have improved greatly the cost per computational cycle continues to decrease exponentially [57].

These circumstances have changed and it has allowed for the experiments explored and discussed further in thesis.

CHAPTER THREE

A SYSTEMATIC REVIEW OF SELF-HEALING SYSTEMS

This chapter explores in greater detail the commonalities between self-healing systems by discussing their backgrounds, contextual uses, and behavioural mechanisms. It provides a point of origin for understanding existing problems in fault detection within these systems, and then outlines and motivates problems within the field before concluding with a synopsis:

Although progress has been made in furthering the autonomy of self-managing systems, implementations across most environments are still largely supervised. In order to fully realise the benefits these systems can provide, a shift to unsupervised learning should be explored.

3.1 Methodology

The assessment of prior self-healing approaches was based loosely on existing systematic literature reviews [58, 59, 60]. Research questions, search processes, quality assessment of output, and how data collection and analysis occurred were sampled before interpreting general results. In short, most self-healing systems were discovered to commonly have properties in three major areas – a topic discussed in Section 3.2 – but only a few provided working prototypes. In order to adhere to an evidence-based approach – such as the one described by

Kitchenham, et al [61] – only those systems that provided evidence of a working prototype were studied in greater detail. The reasons for this will become more apparent later in this section.

3.1.1 Search Process

Discovery of associated literature happened through standard manual search of journals and conference papers through a variety of online academic search engines. Major surveys were evaluated through journals such as the *Journal of Systems and Software*, *Transactions on Information Theory*, *International Journal of Adaptive Control and Signal Processing*, *Decision Support Systems*, and *Transactions on Autonomous and Adaptive Systems*, amongst others. More recent and cutting edge approaches were taken from conference-based publications.

The largest and most respected conferences were sampled for papers that covered the topic of fault detection, identification, and self-healing systems including the *International Conference on Autonomic Computing* (ICAC), the *International Conference on Cloud and Autonomic Computing* (CAC), the *International Symposium on Cluster, Grid and Cloud Computing* (CCGrid), the *International Conference on Autonomic and Autonomous Systems* (ICAS), the *International Symposium on Software Engineering for Adaptive and Self-Managing Systems* (SEAMS), and the *International Workshop on Adaptive Self-tuning Computing Systems* (ADAPT).

From both types of publications, references were taken for further literature to review and then associated into basic categories including: relevant studies, research trends, technique evaluation (both practical and theoretical). Additionally, resources were enquired about directly from academic colleagues at the University of St Andrews believed to be familiar in the subject matter based on their publications.

In total, the initial review focused on some 80 accredited research studies. This expanded greatly when neighbouring topics such as theories behind learning algorithms and other aspects of machine learning were taken into consideration. Ultimately, as many as 170 papers were reviewed to varying degrees – some of which early technical reports, and pre-prints. Even some unaccredited publications from arXiv.org from well known researchers were examined (*e.g.* Bengio, et al. [62]).

In order to produce research that was more reliably founded on evidence-based approaches, work that had a practical approach was emphasised. Specifically, publications that claimed to have a working prototype or reproducible experiment were short-listed for review. This collection of 20 papers became the base set of publications for further inference.

3.1.2 Research Questions

Research questions were not immediately generated but rather synthesised based on common exigencies in the papers that were reviewed. During this time it was noticed that many self-healing systems had common properties in their implementation and that it appeared some of these properties were related.

Ultimately, the research questions synthesised were:

1. In what computing environment is the fault generated? (*i.e.* What is the context of the self-healing system implementation?)
2. How are self-healing computing systems being administered from a design perspective? (*e.g.* ad-hoc, centrally, *et cetera*)
3. What degree of human interaction is expected during operation of a self-healing system?
4. What degree of human administration is required to meet with the ultimate intended goal of self-managing systems, and what, if any, methods have been agreed upon?
5. What kinds of faults are being examined in self-healing systems?
6. Are faults being detected accurately in self-healing systems?

   If so,

   a) How quickly is the fault detected?

   b) Given a fault is detected, how accurately is the fault *identified*?

   c) Is fidelity (*i.e.* total number of samples or observations) an issue here and what is its effect on results?

7. Finally, given two or more self-healing approaches with accurate results, what kinds of direct comparisons exist?

This thesis bases its own research and experimental structure upon these questions (see Section 3.1.2). Although not fully addressed, they are the foundation for how the experiments contained herein have been formulated.

3.1.3 Quality Assessment

To ensure quality, a table was created to track and assess properties noticed in self-healing systems prototypes. This included intended operating environment, and how the systems were being managed. Management of systems was divided into two parts: internal logic, and how often human interaction was required. Of the initial 20 publications, two papers did not meet requirements that clearly outlined the use of the software being proposed or they offered vague descriptors of their implementation, and two papers required unique parameters to address some of their operating characteristics (GPAC [39], and OSIRIS-SR [34]). In the latter case, both were ultimately included.

The primary criteria used to select frameworks was based on the presence of:

1. A description of the intended computing environment or operating conditions,
2. A description of how the systems were expected to be instantiated and managed,
3. A claimed working prototype – papers exclusively focused on theory were not evaluated – and,
4. A description of whether or not the system used labelled or unlabelled data when being initialised.

From these criteria, 15 self-healing systems frameworks were evaluated directly. The remaining excluded papers were not directly compared, but were still used for scientific inspiration.

3.1.4 Data Collection & Analysis

Data collected from each publication consisted of:

1. The name of the framework (if given), or the title of the paper in the case of its absence,
2. The authors of the framework and associated collaborators (if appropriate),
3. The year of publication,
4. Which self-* properties, if any, the paper discussed or addressed,
5. Citation details (in BiBTeX),
6. Notes on supervised, semi-supervised, and unsupervised requirements in using the frameworks,

7. The frameworks associated management style (top-down or bottom-up), and
8. The intended computing operating environment (e.g. peer-to-peer (P2P), cloud, grid, etc).

For items 4–6, frameworks could meet more than one criteria as some of these properties occurred more than once. For example, labelling requirements can be different within certain components of the same framework [27], and ad-hoc computing behaviours can take place between systems designed to be centrally managed [37].

The collected information was reviewed by colleagues and peers both for validity and general scientific interest. Reviews were subject to available time and resource constraints and although are not proscribed standard practice were provided anyway.

3.1.5 Results

The initial survey revealed 12 instances of supervised learning, and 6 semi-supervised and 4 unsupervised learning approaches, respectively. Studies were divided between 8 bottom-up and 12 top-down approaches – this would later even to 12 each during the development time of this thesis. Eleven frameworks were assessed to be able to operate in standard, n-Tier environments; 16 were able to operate in clouds. Nine showed the ability to work in grid or grid-like environments – with many overlapping other possibilities. Only one study was exclusively evaluated as operating in a grid [63].

Throughout the course of this thesis, information surveyed evolved and, of course, the published worked produced and included in this study added a number of factors. These details are further explored in the subsequent sections, but a detailed synthesis of the collected, total results can be found in Section 3.3, and in table 3.4.

3.2 A Comparison of Self-Healing Systems

Self-healing frameworks leverage a diverse set of approaches to autonomously detect, identify, and recover from faults. This section discusses and compares self-healing approaches based on three primary aspects: *management style*, *computing environment*, and *learning methodologies*. These aspects are often interrelated and can play an important role in determining the

effectiveness of a given self-healing solution. As such, some self-healing approaches have been implemented more commonly under specific management styles and computing environments.

The following subsections are organised as follows: Section 3.2.1 contrasts top-down and bottom-up management styles that utilise self-healing frameworks. Section 3.2.2 discusses computing environments, and contrasts different self-healing behaviours commonly found within grids, clouds, and standard infrastructures. Lastly, Section 3.2.3 provides an overview of learning methodologies used to autonomously detect and recover from faults. A distinction is made between supervised, semi-supervised, and unsupervised methodologies, and under what circumstances they are most commonly implemented.

3.2.1 Management Styles

Managing complexity in computing environments has led to an abundance of architectural and systems management techniques. This chapter focuses on two specific styles: Top-down, and bottom-up. Top-down approaches organise systems into hierarchies by leveraging authoritative nodes. These nodes control, propagate, and validate the behaviours of subordinate child-nodes within the computing environment. Conversely, bottom-up methodologies operate in an *ad hoc* fashion, leveraging neighbouring devices to make or suggest changes to configuration state.

Each style divides computing environments into smaller, more manageable sub-components. The division of systems into sub-components helps to address the natural complexity that arises when managing multiple devices. This includes aspects from change management, divisions in workflows, and enacting policies to automate systems tasks. Depending on the management style, however, the nature of the sub-components also changes to provide different advantages and disadvantages. It is often the case that management styles are selected based on computing environment specific needs – a subject discussed further in Section 3.2.2.

Top-down Management Styles

Top-down management styles are based on a hierarchical infrastructure for accepting and enacting policies on child systems [64]. This is often realised through the use of databases on parent-nodes, which subordinate nodes periodically communicate. By changing information within these databases, the collective behaviour of systems communicating with the parent can be altered. Thus, rather than requiring an administrator to access each system individually, top-down methodologies can execute instructions autonomously.

Rainbow [65, 66] is a self-healing framework that leverages a centralised, top-down management style. Utilising a set of *system concerns*, child-nodes are divided into clusters based on a similar set of expected behaviours. These properties are collectively described as system *roles*, and are maintained by a single Rainbow instance. An administrator then provides a set of constraints and recovery plans, which the service uses to evaluate systems behaviour. Evaluations occurred using a three-tiered, abstract architectural model that autonomously categorises systems behaviours. If a fault is detected, the server's configuration is then altered using recovery plans associated with the system's synthesised role, and respective constraint model.

Rainbow's approach to dynamic systems evaluation, and its centralised methods, are arguably foundational by many subsequent approaches. This includes the ability to utilise centrally located recovery plans that are associated with the identification of specific faults [67], and the use of recovery plans that have been created by systems administrators at run-time. Once enacted these results are stored for later use within a centralised database – a technique sometimes referred to as case-based reasoning (CBR).

Localising configuration changes to a single point has the benefit of reducing human error during implementation, and retaining a homogeneous configuration baseline within a computing environment. Top-down management styles are useful in ensuring predictable recovery behaviour, and are widely utilised [65, 68, 69, 67]. Conversely, centralised infrastructures often require extensive pre-configuration and training before they can exhibit self-healing behaviour.

MARKS+ [68] leverages a comparable approach to Rainbow by using what it refers to as *healing manager* nodes to select and implement pre-defined recovery plans. The recovery plans are again evaluated based on a constraint model, but also include a *service availability mapping*. This mapping, combined with a collection of behavioural unit tests, provides context to the evaluation of the constraint model. Systems determined to be in a faulty state are removed from service until a 'good' behavioural context can be re-established via the the return of the system to a previously known working configuration or *state*. For MARKS+, healing managers facilitate these behaviours by acting as a centralised orchestration service. This is similar to Rainbow in that both approaches use an architectural perspective to facilitate resource discovery and recovery behaviours.

The use of *behavioural skeletons* is another perspective on understanding systems activities in top-down infrastructures [69]. Behavioural skeletons are similar to models and consist of an abstract collection of patterns that can be used to evaluate a system's behavioural properties. When combined with a set of constraints, or *contract* [69], top-down styles can attribute context

to systems behaviours without depending on pre-defined roles. This has the advantage of not requiring developers to commit to pre-approved configuration states. Similarly, skeletons and contracts can be used to provision a specific subset of information to child-nodes – such as configuration data or faults. Whilst the child-nodes retain this information locally, a reduction in the need for 'call-backs' to management services remains present. This allows the systems to work more independently and utilise external resources only when required.

The use of locally provisioned self-healing logic is similar to the two previous approaches in that it leverages rule-based action policies to decide on recovery strategies. However, it differs in how systems are allowed to interact, and provides an approach for leveraging more autonomous behaviours. The latter is an artefact that has been extended in subsequent publications [38]. Rather than using a series of contracts, SASSY handles infrastructure management through the use of dynamic model generation called *Service Activity Schemas* (SAS's).

By aggregating these SAS's, an architecture can be dynamically mapped into subgraphs. This allows not only the systems to be modeled individually, but the service architecture itself to be evaluated in a dynamic fashion. Consequently, using this approach affords greater flexibility in compartmentalising faults within the environment than other top-down frameworks, and provides more effective management of resources than stand-alone top-down service discovery methodologies.

MOSES [70] takes a similar approach to SASSY in that management of the service architecture itself is leveraged in detecting and recovering faulty systems components. Like SASSY, MOSES dynamically models the architecture in which it is operating. By using a *position manager* this framework determines if the service's detected resources can be combined into a usable model. Once completed, an *adaptation manager* addresses any faults or quality of service issues encountered by using a series of vectors abstracted from the services model. This information is then abstracted into an ordered list of service priorities that can then be used to direct or redirect service flows – even in the presence of conflicts.

The sampled centralised management styles exhibit similar self-healing logic when recovering from faults. In most instances, the use of behavioural testing is implemented with a contextual reference – such as a constraint or systems model. This is further expanded upon by user validation in the case of supervised methodologies, or by using predictive measures to synthesise recovery solutions.

Furthermore, the use of these techniques in a centralised orchestration service affords many benefits – including the ability to retain control of the infrastructure from singular management points, and being able to leverage re-use of recovery strategies. This is seen most readily in

self-healing systems where the frameworks are given models at their instantiation, including SASSY [38], RAINBOW [65, 66], and other techniques [67, 69, 68]. This is in contrast to systems that inherit or infer self-healing behaviours; a topic discussed further in 3.2.1. Learning methodologies for each management style are discussed further in 3.2.3.

Bottom-up Management Styles

Bottom-up management styles emphasise *ad hoc* interaction between systems. Systems within these environments typically infer self-healing behaviours based on independent sampling, either of the service infrastructure at large or neighbouring systems, and exhibit a greater degree of administrative autonomy. They represent a direct alternative to approaches that leverage centralised management, and typically demonstrate more exploratory behaviours. This type of systems management can require less initial configuration than centrally managed approaches, but at the cost of predictability and individualised control.

Although *ad hoc* systems management comes in a variety of forms, this survey focuses on three distinct approaches: system-to-system [32, 34, 71], localised healing [37, 36, 28, 24, 27, 72], and those that utilise atomic interfaces to synthesise virtual resources [39]. These approaches were selected based on commonalities observed in the sampled papers.

System-to-system frameworks are capable of making changes by sampling from or delegating to neighbouring nodes. This is contrasted by localised healing frameworks which avoid administering other devices, and use information obtained from neighbouring systems to self-elect behavioural modifications. Atomic frameworks exist as a hybrid of these two approaches by exposing their resources in an non-holistic, read-only fashion. They can either self-elect or suggest changes to external devices, or directly access external resources as if they were locally present.

In a system-to-system infrastructure authoritative actions are delegated dynamically through the analysis of environmental knowledge. Examples include frameworks that observe both the performance and service availability of neighbouring devices [36, 37, 34]. In the case of Embryo-ware [37], a set of administrator supplied configurations provides each system with the ability to autonomously adapt from a ‘totipotent’ state – *i.e.* a neutral configuration – into one of several pre-specified roles. This behaviour is initiated based on each system’s local perception of the over-all performance and relative needs of the service infrastructure. If a service has reached a capacity threshold for its front-end web-services, for example, and the system has a totipotent configuration, it can dynamically adopt a web-role and join the front-end pool to

increase capacity. Once the service has been evaluated as no longer needing additional front-end resources it then reverts back to its neutral state.

By treating systems as modular components, Embryo-ware addresses a key problem present in *ad hoc* infrastructures – drift in baseline systems configuration. As systems continue to operate they naturally encounter events that create unique systems configurations and states. This can create scenarios where systems are difficult to predict and can reduce the effectiveness of existing self-healing behaviours. By resetting the local system's state to pre-defined known-working configurations, divergence in systems operations is dramatically reduced. This allows for techniques that depend on assumptions related to the systems behaviour to continue to be effective well after initial deployment. It also allows for servers to be treated as dynamic resources within the service architecture and to transparently address the workloads associated with predefined groups of individual service components.

Transparently updating service components is an approach also used by OSIRIS-SR – an extension of OSIRIS [32] and Chord [33]. However, unlike Embryo-ware, OSIRIS-SR uses a transitive management service to create 'supervisor nodes' that facilitate self-healing behaviours. These nodes leverage a distributed hash-table to establish service parity, and to facilitate work delegation of a given resource. This allows service availability to be preserved even in infrastructures with high rates of churn, and for systems to orchestrate service flows whilst addressing faults–all without a centralised infrastructure.

Rather than shifting a system's role or instantiating a supervisory service, systems within the computing environment may also have the ability to assign work directly to each other [71]. VieCure utilises an activity management service to understand local and remote service state [71]. Like Embryo-ware, this framework is installed locally on each system, and configured by a set of policies that guide self-healing behaviours. The policies combine 'interaction patterns' and constraints into a *behaviour registry* – a dictionary-like object capable of recognisable systems states that are used to indicate when self-healing behaviours are required. If a constraint violation occurs, the system can either choose to heal or delegate work to a neighbouring node. VieCure, OSIRIS-SR, and Embryo-ware operate holistically. The expression of their self-healing logic is based on the evaluation of their respective computing environments as a whole. However, not all *ad hoc* frameworks operate in this fashion.

Atomistic perspectives, such as the General Purpose Autonomic Computing framework (GPAC) [39], view and evaluate systems' resources as individual components based on 'resource definition policies' that are supplied by an administrator. The benefit of atomising components is that they are usable remotely by other systems. To accomplish this, GPAC first

builds a model of local systems operations by utilising a four stage control loop similar to MAPE+*K*. The model is populated by querying either a remote or locally running service that discovers resources. Discovered resources are then integrated with the model information by a policy engine to create the aforementioned *resource definition policy*. This allows resources to be directly accessed, regardless of physical location.

Sharing resources leads to a natural integration between systems, and illustrates a perspective for mitigating faults remotely in a bottom-up fashion. It also highlights the primary caveat that exists between other approaches in that systems must be able to accept changes to their configurations from neighbouring nodes. In some computing environments this property is undesirable – particularly when the trustworthiness of other nodes is an unknown. For these cases, localised healing strategies are the preferred methodology.

Localised healing frameworks avoid directly administering other devices. Instead, each framework instance is exclusively responsible for its local system's health, resources, and configuration state. This includes determining when issues are caused by local or external factors. Localised faults are mitigated in a similar fashion as other frameworks. A set of constraints and policies are provided by administrators which the systems use to detect and recover from faults. However, faults determined to be external to the system are addressed differently. External faults are either ignored, referred to another system, or, if possible, mitigated locally. These approaches are not designed to address the source of the error, but to maximise the availability and performance constraints of the computing environment – often within predefined guidelines.

For example, lowering the fidelity of content being served by front-end web-servers is one way to meet performance constraints [35]. If a server cannot deliver content at the rate expected – *e.g.* due to too many concurrent connections – it can elect to reduce the volume of data sent for subsequent data requests. This approach does not directly address the state of other systems, but instead focuses on those issues that can be resolved locally. Frameworks that focus on localised self-healing techniques often use *roles* to facilitate the re-use of self-healing logic and to meet constraints [35, 24]. This is particularly useful in self-healing systems that operate within a single tier of a computing environment.

WS-DIAMOND [24] is a localised healing framework specifically developed for front-end web-services. It uses two concurrent control loops to diagnose and recover from faults. The 'inner' control loop focuses on the mitigation of faults that prohibit basic systems operations. This can include resources that are critical to the system's role, and the state of services. The outer control loop addresses issues related to quality of service (QoS). If a system is not capable of

performing within a set of constraints, an error is raised that the outer control loop attempts to mitigate. Other frameworks have mimicked the QoS approach, but sans the use of multi-tiered control loops [73]. However, the basic approach used in these systems are essentially identical. Each failure instance is treated as a separate case from which to analyse the results of systems configuration tests. This allows faults to be categorised based on the systems role, and located using differential analysis of the systems configuration data.

Determining the source of an error is a non-trivial process. Systems configurations are complex sets of information, and often contain relationships between features and properties that are not easily classifiable. Dynamic systems modelling represents one approach for understanding correlations between faults and configuration state. In localised healing frameworks, such as Plato [27], Unsupervised Behavioural Learning (UBL) [28], FDFs [18, 19], and Shadows [72], these approaches have been used to categorise and compare the state of a system with historical information, such as systems configuration or performance data – a topic discussed further in the following Section, 3.2.3.

The bottom-up management of UBL, the FDFs, Plato, and Shadows all follow in the footsteps of top-down frameworks, such as Rainbow, that utilise architectural modeling techniques at system run-time. However, these frameworks all leverage a set of operating constraints that allow for differences to be discovered between systems behaviours, and recovery methodologies to be synthesised, rather than requiring them to be applied by administrators.

Recovery strategies for these frameworks operate differently which in turn has an impact on how they are managed. Plato utilises genetic algorithms to search for optimal systems configurations and enacts recovery methodology via reconfiguration. The results of each configuration undergo a differential analysis that examines the health and performance of various systems models before implementation is done independently.

Shadows uses a model repository to determine a recovery strategy. The repository is populated via two mechanisms - a code extraction methodology, and a CBR-based approach similar to those described by Carzaniga, et al. [74], Cheng, et al. [65], and Hassan, et al. [75]. However, rather than requiring administrators to update the repository manually, Shadows automatically builds role-based recovery solutions without human intervention. This is accomplished by using a combination of statistical and predictive modelling to synthesise configurations and evaluate potential solutions to detected faults before (re)-implementation. Once a solution has been found it can be validated and shared throughout the environment where behaviours are determined to be similar. This unique use of case-based reasoning allows the framework to leverage the advantages of *ad hoc* systems management without depending on centralised

infrastructure or human administrator to approve new recovery methodologies. By removing the supervision requirement of this CBR approach anomalies can be detected that were not previously known.

UBL operates a little differently than the previous mentioned bottom-up approaches in that it has some hybrid properties. Specifically, it uses a centralised training virtual machine (VM) to help independently evaluate solutions by populating and instantiating a SOM. Additionally, it has the capability to resolve problems locally without this centralised component. This technique allows systems to build their own recovery solutions at run-time by leveraging a vector based approach for aggregating systems configuration and performance data. Once the information has been obtained it is then classified and subsequently analysed (*i.e.* mapped), faults are then inferred through a differential analysis of changes in both behaviour and configuration state of the system in question similar to the aforementioned bottom-up strategies. This framework is the most similar in approach to the experiments described within this thesis – further details on how it operates are discussed at length in subsequent Chapters.

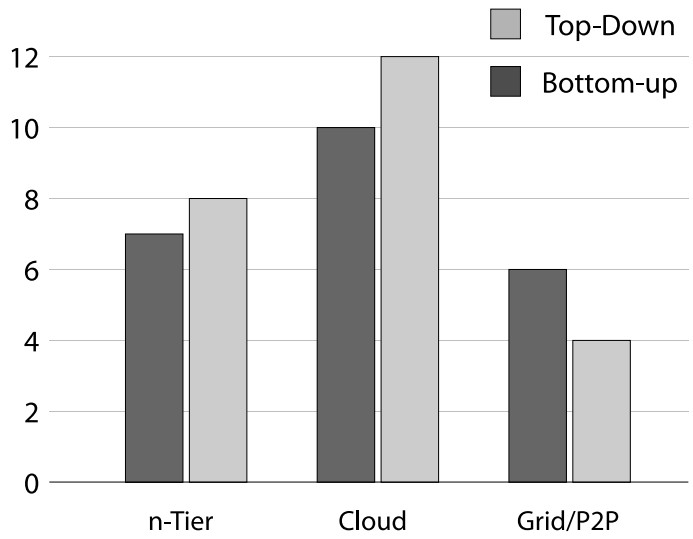

Figure 3.1: Management Styles versus Computing Environments. Managed environments – such as Cloud and n-Tier infrastructures – show a preference for top-down management styles, whereas ad hoc computing environments prefer bottom-up management styles.

The management style of a self-healing framework is often related to its computing environment (Figure 3.1). In the case of *ad hoc* systems administration, the behaviours exhibited are inherently less predictable than those that leverage centralised methodologies. This comes

as a caveat of allowing systems the ability to independently explore solutions outwith those having been directly supplied. Specifically, systems that leverage a bottom-up management style appear to be more prone to use semi-supervised and unsupervised learning techniques to achieve dynamic recovery solutions. While this approach is by definition more autonomous, it does not necessarily mean that it is more usable. Some environments may be required to use only specified recovery solutions to address specific service aspects – such as risk management or high availability requirements. In such cases solutions such as Embryo-ware may be better suited than UBL, Plato, or Shadows.

Choosing a management style for a self-healing framework is a multi-faceted problem and can depend on a number of extraneous factors – such as the environment in which the system is intended to operate, or acceptable levels of downtime or resource usage in searching for new solutions. The attributes between environments and learning methodologies, and associated self-healing behaviours, are discussed further in Sections 3.2.2 and 3.2.3, respectively.

3.2.2 Computing Environments

Computing environments are collections of resources used to manage and facilitate a given set of systems. Depending on the needs of the systems, computing environments can have different infrastructures and assets. This survey focuses on three types of infrastructures: Standard, virtualised, and *ad hoc*. Each infrastructure type presents differences in how self-healing frameworks access, categorise, and utilise resources. These differences can have profound impacts on the approaches used by self-healing frameworks and their respective goals.

Standard infrastructures typically comprise three categories when discussing systems responsibilities: Front-end, middleware, and back-end. Front-end systems are responsible for establishing and maintaining connections to clients, middleware provides facilitating services such as encapsulation, transport, or orchestration, and back-end systems are responsible for the provisioning, storage, and parsing of information. This division of responsibility is the basis for establishing reusable code in many self-healing frameworks – regardless of computing environment – and promotes scalability by organising systems into reusable, interchangeable components. This allows for extensibility in behaviours and interchangeability of failed devices.

Virtualised infrastructures emulate physical assets by using multi-system resource management techniques. Instead of building a physical machine with a specific role, resources are dynamically allocated from a collection of physical machines to build virtualised ‘instances’. These instances operate in the same fashion as physical systems. However, as the hardware

itself is a software manifestation, 'physical' changes can occur more rapidly and in a more autonomous fashion than in standard infrastructures. In addition to rapid reconfiguration, virtualised infrastructures handle change control exceptionally well. This is primarily due to the use of systems clones (*i.e. images*) when instantiating new instances. Images allow for quick replacement, re-provisioning of faulty systems, and fast comparisons between systems' configurations. These properties make virtualised infrastructures heavily leveraged in cloud computing environments – a trend that has been increasing in recent years (figure 3.4).

Standard and virtualised infrastructures share several key properties. They are often owned or operated by a single entity, have low rates of churn, and typically leverage centralised management styles (Section 3.2.1). These aspects are vital in meeting established minimum operational requirements such as availability, reliability, and performance expectations – sometimes referred to as service-level agreements (SLAs). However, there are computing environments that do not share or require these properties. In these cases self-healing frameworks leverage *ad hoc* infrastructures.

Ad hoc infrastructures are unique from other approaches in that systems membership is voluntary. This property is related to *ad hoc* management styles, which enable systems to act as an authoritative point and evaluate its infrastructure independently – sometimes referred to as *self-elected behaviours* – but is different in that it refers to the association a system has to a specific environment. The ability for systems to join and leave an infrastructure has advantages in that they are better suited for some distributed computing uses, and can potentially operate at lower costs. The transient nature of *ad hoc* infrastructures pose unique challenges for self-healing frameworks. Notable examples include higher rates of churn [11], issues with reputation [76], security [77, 78], multi-party administration [63, 79], and a lack of baseline configurations between systems [31], amongst others.

Computing environments are sometimes comprised of multiple infrastructure types. Some environments, for example, may have systems that are capable of interacting with each other in an *ad hoc* fashion, but may also depend on a centralised service model [72, 37].

In most cases self-healing frameworks have been developed to meet specific needs within a single tier of an infrastructure – such as a front-end web-service [35, 24, 73, 18, 19, 67]. Nearly all self-healing frameworks that are designed to operate within a single tier are capable of being implemented in a virtual infrastructure. However, not all self-healing frameworks are restricted to one area of responsibility [65, 37]. The most common tier-specific self-healing frameworks are those that focus on front-end systems [35, 74, 24, 67].

Systems that approach front-end web-services utilise a variety of approaches, including multi-

tiered control loops [24], fidelity reduction [35], and behavioural modeling [67], amongst others. These self-healing frameworks can promote an intermediate stage for adapting existing infrastructures towards stronger administrative automation.

Each system in a standard infrastructure must be maintained individually. This has several notable consequences including increased provisioning times, the potential for inconsistencies in configuration and implementation, and a natural deviation in configuration baselines over-time. These problems have been partially addressed by self-healing frameworks through the use of CBR and centralised management styles [72, 69, 68, 67, 71]. Centralised approaches leverage an often human supplied correlation between root causes of faults and their respective recovery strategies. As the expected outcome is based upon assumptions of previous state, these approaches can become less effective as configurations diverge. As changes occur within separate infrastructures outside of the control of the framework, this problem becomes more complex.

Virtual infrastructures help to address baseline configuration deviations, dynamically provision new resources, minimise the impact of external infrastructure changes, and improve deployment and recovery times. The majority of these advantages stem from the use of images which, as previously mentioned, help to maintain standard configurations between systems. Virtual infrastructures also come with several major disadvantages, the largest being cost to operate, proprietary standards for larger implementations, and challenges for physical expansion. However, virtual infrastructures provide useful properties to frameworks that use tier-based and search-space approaches to resolving faults [72, 27].

Frameworks that leverage search space methodologies require one of two conditions to occur before executing self-healing behaviours: Either an acceptable solution must be converged upon, or all available resources are exhausted. In the latter case, the framework picks the best solution found [28, 80, 27]. Standard environments limit the availability of resources to the physical capabilities of the system upon which the framework is instantiated. Virtual environments provide an advantage by allocating resources beyond the immediate instance. This promotes the self-healing behaviours from break-fix objectives to optimisation strategies (e.g. [81, 27, 70]).

In addition to optimisation, the dynamic allocation of resources is useful for promoting stabilisation in computing environments. There are several self-healing approaches that explore stabilisation in standard environments including dynamic role-adoption [34, 37], resource discovery [65, 66], resource policies, atomisation [39], and reduction in content fidelity [35]. In some cases, virtual infrastructures demonstrate comparable advantages by using instancing.

Embro-ware's ability to use 'totipotent' systems to shift to and from needed roles is comparable to virtual infrastructures' ability to dynamically spawn new server instances – assuming an image exists for the needed role and a feedback mechanism is actively monitoring service state. Both approaches represent a way to preserve QoS in an environment, and minimise the need to reduce content fidelity.

Virtualisation universally addresses a major advantage of Embryo-ware: The ability to use a single subset of resources to address multiple roles within a service or computing environment. This concept is difficult to implement in standard, multi-tier infrastructures. Systems that are organised into tiers have external considerations when communicating with other devices. This includes networking configurations, security measures, and other exigencies of a practical nature that are outwith the control of the framework. With standard and virtualised infrastructures, the barriers between tiers are often preserved. One approach for avoiding these issues is to treat the computing environment as a ring [32, 33, 34]. However, it is worth noting this effectively converts the standard tier-based environment into an *ad hoc* infrastructure.

Ad hoc infrastructures avoid many of the organisational requirements of standard and virtual infrastructures. In ring-based approaches, systems are often required to accept a centralised point of management, and be operated within a confined set of conditions, such as a specific configuration or role. In *ad hoc* infrastructures systems are defined by their ability to carry divergent configurations and self-elect behavioural changes and states. These properties help to mitigate security issues, high rates of churn, diversity in systems configuration, and multi-party administration. Although this chapter covers no frameworks that have been explicitly designed for entirely *ad hoc* infrastructures, several approaches expect and utilise *ad hoc* self-healing behaviours [28, 34, 37, 27].

These behaviours range from self-electing systems roles [37, 34], to aggregating resources between systems [39]. In the former case, each system evaluates the state of the service independently by querying neighbouring devices. If a system chooses to adopt a new role or configuration, it is ultimately centrally managed as the pre-specified roles must be provided to each individual system before they can operate. However, the collective behaviour of each individual system evaluating the service demonstrates an emergent approach to managing the infrastructure health. Experiments with biologically inspired paradigms [82] further suggests that gradients, fields and other "spatial" structures [83] can offer robust adaptation to local challenges and failures, and can act as a programming platform on which to construct complex applications.

Plato [27], UBL [28] and the FDFs [18, 19] demonstrate this perspective by leveraging

systems that can holistically self-evaluate service state using biologically inspired computational approaches. These approaches have distinct advantages in that systems need not be provided with pre-specified recovery strategies, and are specifically designed to exhibit self-adaptive processes through continuous environmental analysis. This affords systems using these frameworks a better suitability towards environments where *ad hoc* management styles and infrastructures are in place–such as the ability for systems to integrate their own self-healing logic by using search-space methodologies. These approaches include, chiefly, Genetic Algorithms (GAs) [36, 27], ANNs [18, 28], HMMs [18], and RBMs [18, 19]. However, search and probabilistic methodologies lack the stable, predictable nature of approaches that leverage periodic human intervention.

Computing environments and the services they house are interrelated. Systems that have the ability to operate holistically require different supporting resources than those that operate in an atomistic [39] or centralised fashion [40]. The self-adaptive behaviours of systems leveraging *ad hoc* methodologies appear to be more advanced with respect to self-autonomy than other approaches. Evidence for this claim is best seen by dividing then comparing the behaviours of systems that leverage either holistic, self-elected behavioural changes or CBR and other typically supervised learning methodologies (Figure 3.2).

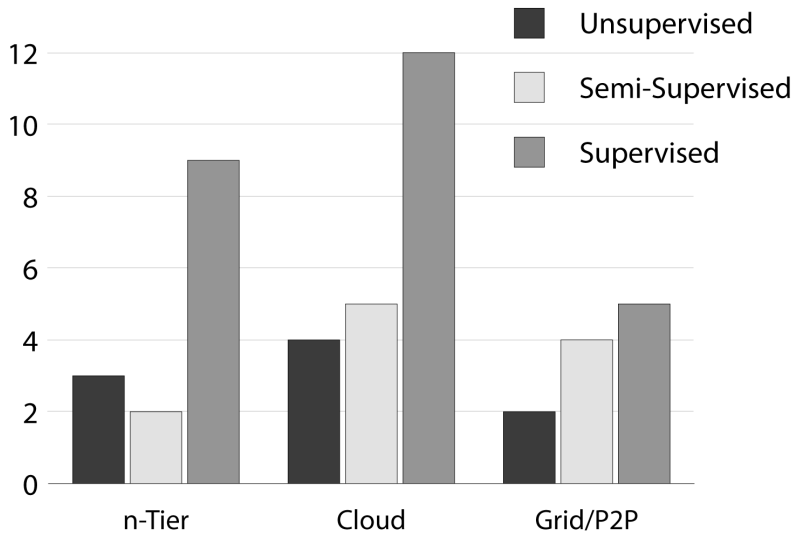

Figure 3.2: Learning Methodologies versus Computing Environments. Most self-healing systems prefer a supervised learning methodology, regardless of the environment it is implemented in. This is useful for ensuring correct behaviours, but also a limitation in potential for autonomy.

3.2.3 Learning Methodologies

Some self-healing systems frameworks rely on heuristic algorithms to correct or change behaviour without human intervention. In order to maximise the effectiveness of these algorithms, learning methodologies have been developed that optimise when and how instructions are executed. These methodologies often utilise recursive, evolutionary, or close-control loop programming techniques to improve and evaluate behaviours. In the majority of cases a feedback mechanism determines both the validity and efficiency of a specific solution. The degree of required human interaction within a feedback mechanism is sometimes referred to as its degree of *supervision* due to the labelling of training information either by a machine or by a human.

Self-healing frameworks can be broadly categorised as being fully supervised, semi-supervised, or unsupervised. Traditional definitions of these terms usually emphasise when or how a system classifies its learned behaviour – either manually, or dynamically – and whether or not data utilised by a specific algorithm has been labelled. As self-healing frameworks can implement multiple learning methodologies – sometimes at varying points within their process structures – cataloguing a framework's learning taxonomy into a single category is challenging. Each implementation must be evaluated independently before an overall assessment can be made about that framework's capabilities. To make matters more complicated, some approaches are not reproducible or their methods entirely disclosed; this restricts the evaluation of their capabilities to theoretical measures, only.

The most common approach to self-healing systems is to use a fully supervised methodology [70, 32, 34, 73, 38, 69, 67, 65, 68]. Supervised methodologies usually require frequent interaction, and extend their self-healing behaviours only upon human intervention. This allows for validated, controlled configuration updates and provides the least amount of uncertainty in systems behaviours [84, 85, 86].

The most frequent implementations of supervised learning are CBR methodologies (Figure 3.2). CBR typically utilises a database of prescribed recovery plans that are correlated to specific faults or events. When a system encounters an error it queries the database for recovery instructions. In those circumstances where recovery instructions have not been previously included, the framework will ask an administrator for a solution, or refer to a default set of actions. CBR approaches extend their behaviour by storing these additional solutions in their databases. Typically, the requirement of human supervision as a required part of the self-healing logic produces the natural caveat of only partial automation.

Rainbow [65] takes supervised methodologies a step further by leveraging dynamic resource discovery with prescribed, role-based recovery logic. Using this approach computing environments are divided into recognisable components that can be used to dynamically build an architectural model of the service infrastructure. Using this model systems and services are categorised within a specified role or type, whilst the architectural model continues to choreograph service interaction and defines expected behaviours. These components are provided by developers before deployment. Once errors are detected, they can be mitigated using the architectural model to restore the service to a known working state or, if unsuccessful, an administrator can update the model at run-time.

WS-DIAMOND [24] and GPAC [39] take similar approaches to Rainbow in that a model is specified to which a system's performance is evaluated. However, rather than monitoring an entire service, each system is managed independently. As previously mentioned, WS-DIAMOND does this by instantiating two concurrent control loops to monitor and correct systems behaviours. Dividing the recovery logic into separate components allows the framework to prioritise and isolate recovery strategies. This is naturally conducive to goal and utility policy implementation within the specified model. A number of extensions to this framework have seen improvements to its detection and recovery logic[1] including the ability to monitor workflows, orchestrations, and choreographies.

GPAC contrasts this approach by utilising *resource-definition policies* to autonomously discover and atomise – *i.e.* individually expose – systems' components into network accessible objects. This non-holistic approach allows the framework to access resources on remote systems as if they were locally present. When combined with a model of the service, systems can transparently heal and optimise the service architecture in a semi-supervised or potentially unsupervised fashion. These policies can also be used to tier service performance based on priority of behaviours or resources.

Performance tiering is a self-healing methodology used to divide systems and service health into levels [72, 35]. These levels in turn are used to understand QoS changes and instantiate behaviours that maximise the usage available resources. Arguably, the most direct approach to defining service levels is to use statically assigned resource constraints. Each level corresponds to a set of QoS metrics or fitness criteria that tells the system when to dynamically reduce content fidelity [35]. Primarily developed for front-end web-services, static service tiering requires a human-supplied policy to determine when content fidelity can either be reduced or increased.

[1] http://wsdiamond.di.unito.it/status.html

In contrast, allowing policies to dynamically set thresholds for self-healing behaviours can have more autonomous results [70, 72, 31, 73, 87]. The Shadows framework, for example, uses a set of SLAs and utility policies to automatically generate behavioural expectations of a system. This allows the system to perform more in line with human-readable goals, such as cost, average service time, and other criteria instead of discrete metrics. It then combines this information with historical performance data to provide internal revalidation of recovery solutions. By using a time windowed mean expectations in behaviour can allow for elasticity versus pre-defined QoS metrics.

In a supervised framework the revalidation of a new set of expectations is normally completed by a member of technical staff. This occurs in a similar fashion as that being leveraged by Shadows: SLAs are compared against a system's overall performance and combined with historical data–such as application logs, configuration files, etc. The addition of correlating events with systems faults provides an advantage in contextually evaluating anomalies [72, 71]. By sampling the system at key intervals, faults can be associated with specific changes and, ideally, their respective sources [18, 19]. This is useful for establishing a root-cause analysis and to map similar events with recovery solutions–sometimes referred to as *Event Driven Monitoring* [28].

Event Driven Monitoring combines a complex set of sensor classification algorithms with run-time analysis techniques for isolating anomalies from normal or established patterns of behaviour. These approaches can range from the reactive use of simple exponential smoothing algorithms in a time series prediction [81], to pro-active prediction of states [88]. VieCure [71] is a CBR-style framework that leverages event detection in addition to direct analysis of metrics. Instead of directly mapping faults to recovery plans, VieCure looks for deviations in expected systems behaviours that can indicate when self-healing is needed.

Events can constitute a series of incidents within a log, or a set of incidents that exhibit either a certain order or rate of occurrence. If an event is determined to coincide with a fault, then a recovery strategy is selected from a known set of working solutions. As expected, unknown events and faults require supervision in the same manner as other CBR frameworks – supervised approaches are limited to reactive fault mitigation as by definition they cannot address what has not yet been observed.

Periodic interaction by administrators remains a caveat of supervised and semi-supervised self-healing frameworks. However, some frameworks have demonstrated an ability to dynamically elect self-healing behaviours without this requirement [87, 28, 36, 37, 26, 27], but the most notable gains have occurred using the aforementioned ANNs [18, 28], HMMs [18], RBMs [19],

and biologically inspired behaviours, such as totipotent adaptation [89, 29, 36, 37] and GAs [26, 27]. Each of these techniques have different properties that relate to their suitability at solving particular tasks – from producing candidate solutions within a given search space [80] to the autonomous classification of sensory information [90]. These approaches range in degree of suitability based on how much risk and resource commitment a specific computing environment or service infrastructure is willing to accept.

Using a GA, Plato can search for and mitigate faults based on correlations between behavioural properties and configuration data. This is a framework that dynamically produces self-healing solutions based on a stochastic search methodology that comprises of multiple candidate solutions [26, 27, 80]. By comparing the operational SLAs and policies with the performance of the candidates individually, a degree of fitness can be ascertained from the candidate. Once the candidates have been evaluated, their individual features are analysed and correlated to produce new candidates. This occurs until either preset resource constraints are met, or an optimal solution is found per the associated fitness functions. In this instance, the utility functions in previous frameworks are analogous to the properties that are emphasised by the fitness functions in genetic algorithms. Each respective function provides the same base purpose: To translate and enact human-readable goals into systems behaviours. Examples of these goals include cost minimisation, application priorities, or performance traits.

This approach allows Plato to stochastically search for and build recovery strategies providing a critical advantage over other methodologies. Rather than requiring prescribed recovery solutions, either during development or run-time, Plato can autonomously produce viable self-healing solutions. However, there is no assurance that an acceptable systems configuration will always be found using this methodology, nor that it will be optimal. This is as expected [27], and inherent to the nature of existing search-space methodologies [80]. Plato's use of GAs is also computationally costly, and can produce behaviours that cannot be anticipated. Thus, a high degree of risk can be associated with this approach.

Complimentary to using GAs, UBL operates by using historical configuration data to autonomously train a SOM [90]. Features in the historical configuration are converted into vectors which are then used as input for predicting behaviour, and feature state. This information helps to analyse the validity and impact on a system's behaviours when configuration changes occur. Once the SOM is trained, the system can then synthesise new, valid systems configurations by predicting which features are causal to specific faults. This approach leverages a smaller search space than the genetic algorithms used in Plato, and consequently presents less risk via potentially divergent systems behaviours.

However, UBL displays some limitations in exploring new configurations and seems to produce a stronger likelihood of local minima in configuration synthesis due to its inability to expire sampled data. This is represented in the purposes of these two approaches being somewhat divergent: The ability to synthesise new, valid systems configurations upon fault, and the prediction of failures within distributed infrastructures. Recently, a comparison of this approach and other stochastic primitives demonstrated an improvement upon these results [20], and combined studies in *feature locality* continue to display positive results [91, 31]. The findings in the former study are the primary focus of this book and are discussed in the remaining chapters.

Separate from either of these approaches is the transparent management of resources within a service infrastructure via dynamic role or service adoption [34, 37]. In each of these approaches systems use information about the general state of the service infrastructure to dynamically elect a localised reconfiguration. However, these approaches differ by allowing systems to dynamically adopt roles through self reconfiguration, in the case of Embryo-ware [37], and the self-instantiation of localised management services [34].

As previously mentioned, these systems are initially instantiated with a representation of the service, a set of roles, and an ability to query service state on remote systems. Using these three components the framework is then able to dynamically adopt new configurations or return to an original, neutral configuration based on service performance. Any device found to be without a base set of configuration data is automatically provisioned with the latest ‘genome’ via a replication agent. This provides a measure of self-configuration and provisioning; a process typically referred to as a separate challenge in Autonomic Computing [3, 5]. The adoption of new roles is facilitated via a differentiation agent that tracks and contextualises roles and expected functions. This agent must then self-elect a role-based on its independent understanding of the state of service.

This approach is contrasted by OSIRIS-SR, a framework that leverages Chord [33] to produce a *Safety Ring* to manage service infrastructures [34]. OSIRIS-SR operates by using supervisory systems roles to monitor and recover from failures in resource availability. These systems leverage meta-data to build an understanding of neighbouring systems behaviours, and then aggregate that information across multiple supervisory nodes. This is similar to Embryo-ware where only neighbouring nodes are monitored and influence the ability of those systems to adopt roles. What makes this approach unique is that any system can elect to become a supervisory node. This is useful for ensuring availability and reliable service management in infrastructures where systems membership can change without notice [10].

The following diagram illustrates trends in the management styles associated with self-healing

frameworks (Figure 3.3.

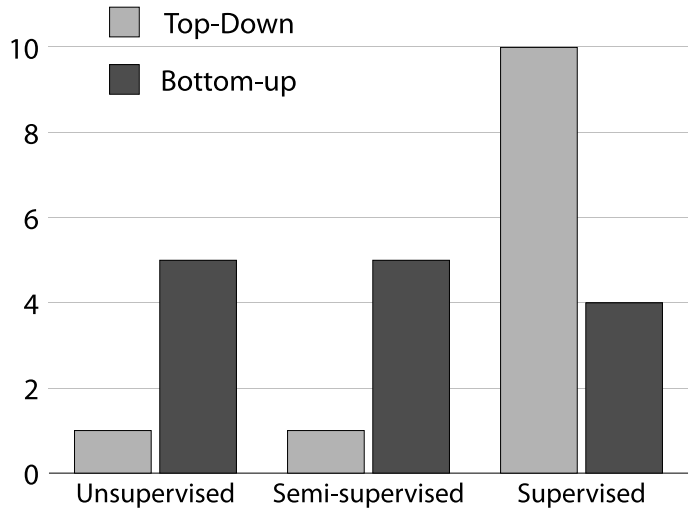

Figure 3.3: Learning Methodologies versus Management Styles. When self-healing system are implemented in a top-down fashion, they tend to leverage supervised learning methodologies. Likewise, bottom-up management styles are more likely to use unsupervised and semi-supervised learning.

There appears to be two key properties immediately evident within figure 3.3: 1.) self-healing frameworks research is driving towards solutions that utilise supervision in centrally managed systems; and 2.) the learning methodology leveraged by the self-healing framework appears to be linked to its management style. Additionally, if we extrapolate this information, it seems that progression towards less supervision is being driven chiefly in ad-hoc computing environments. However, because of the sample size of grid and P2P approaches being relatively low, this may not be immediately evident. Instead, developments in this area appear to be occurring in cloud computing and other environments that leverage virtualisation.

3.3 Synthesis

Self-healing systems methodologies are becoming more autonomous, but remain dependent upon either required periodic human interaction or the acceptance of uncertainty in systems behaviours. This finding comes as self-healing methodologies continue to specialise based on external factors such as their intended computing environment and respective management styles.

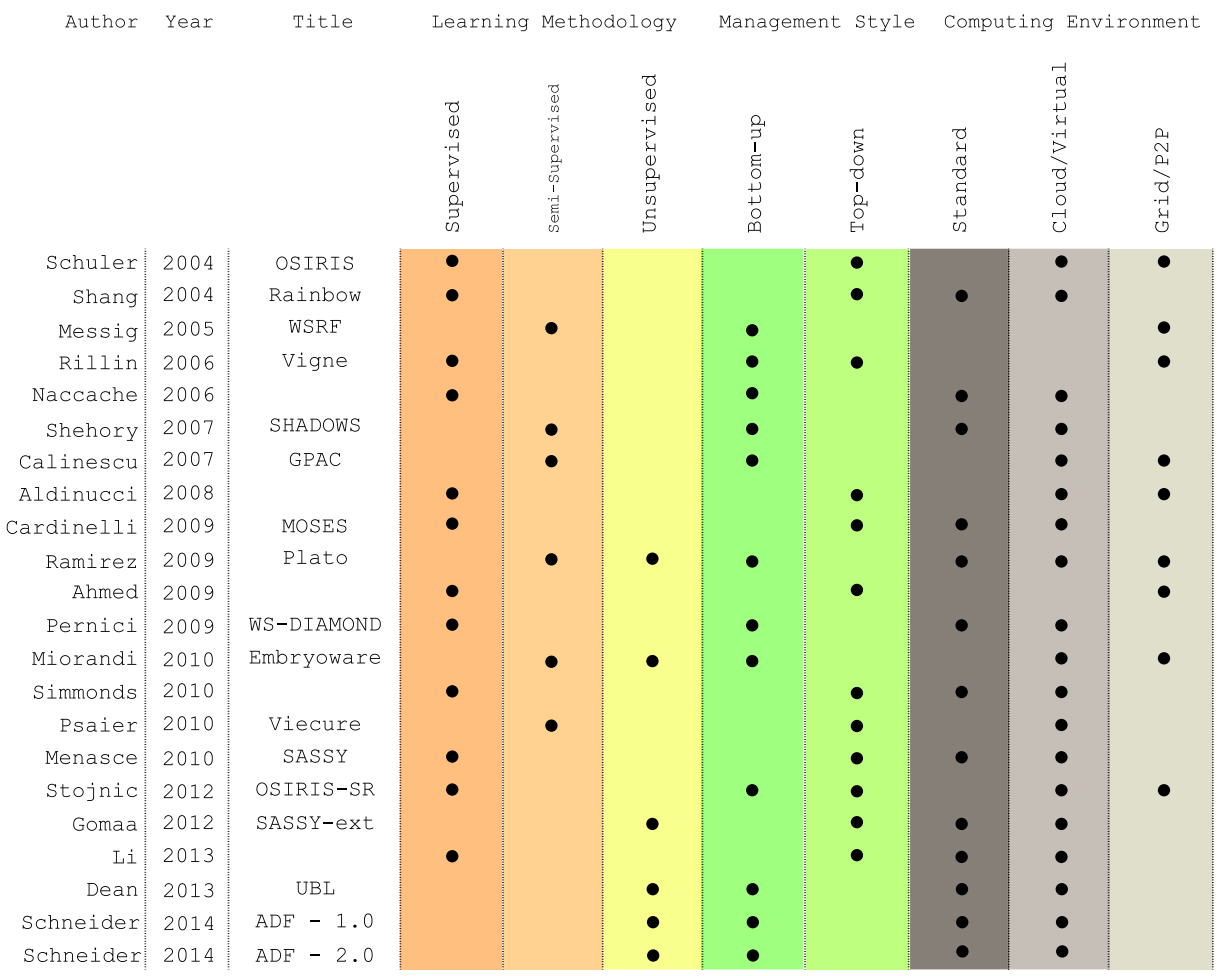

Figure 3.4: Self-Healing Systems Frameworks. Self-healing systems frameworks as categorised by learning methodology, computing environment, and expected management style by first author's last name, year of introduction, and framework title (if appropriate). In some cases frameworks exhibit abilities to operate under multiple assumptions – these incidents are represented by additional bullets within the graph. Figure 2.8 divides this information into percentages with each entry represented once per category.

Notably, a framework's specialisation has been shown to provide distinct advantages in autonomously identifying and resolving faults. These advantages play pivotal roles in understanding how self-healing frameworks are evolving.

Furthermore, many approaches display behaviours that are not universally desirable self-healing approaches are diverging based on their specialisations. This is a concept that until now has not been explicitly addressed within the field. By contrasting where self-healing frameworks are being implemented, an understanding is gained of where self-healing systems are making progress and towards which specific problems.

The intended computing environment of a given framework is a foundational factor in evaluating the success of its self-healing behaviours and has produced a divergence in the types of self-healing systems that are being developed. Environments that require a greater degree of control of their systems often exhibit centralised management techniques [68, 71, 65, 67, 38, 87, 73, 32, 70]. These approaches are evaluated based on how predictable their behaviours are, and often intentionally build in a requirement for human intervention. Conversely, frameworks that operate in *ad hoc* infrastructures [24, 39, 27, 37, 28, 34] are often expected to exhibit behaviours that do not require human intervention, and in some cases to synthesise new self-healing strategies. This result is an artefact of computing environments having inherently different properties, exigencies, and requirements. The result has been that self-healing frameworks have developed specialised strategies that address each of these factors, explicitly (Figure 3.4).

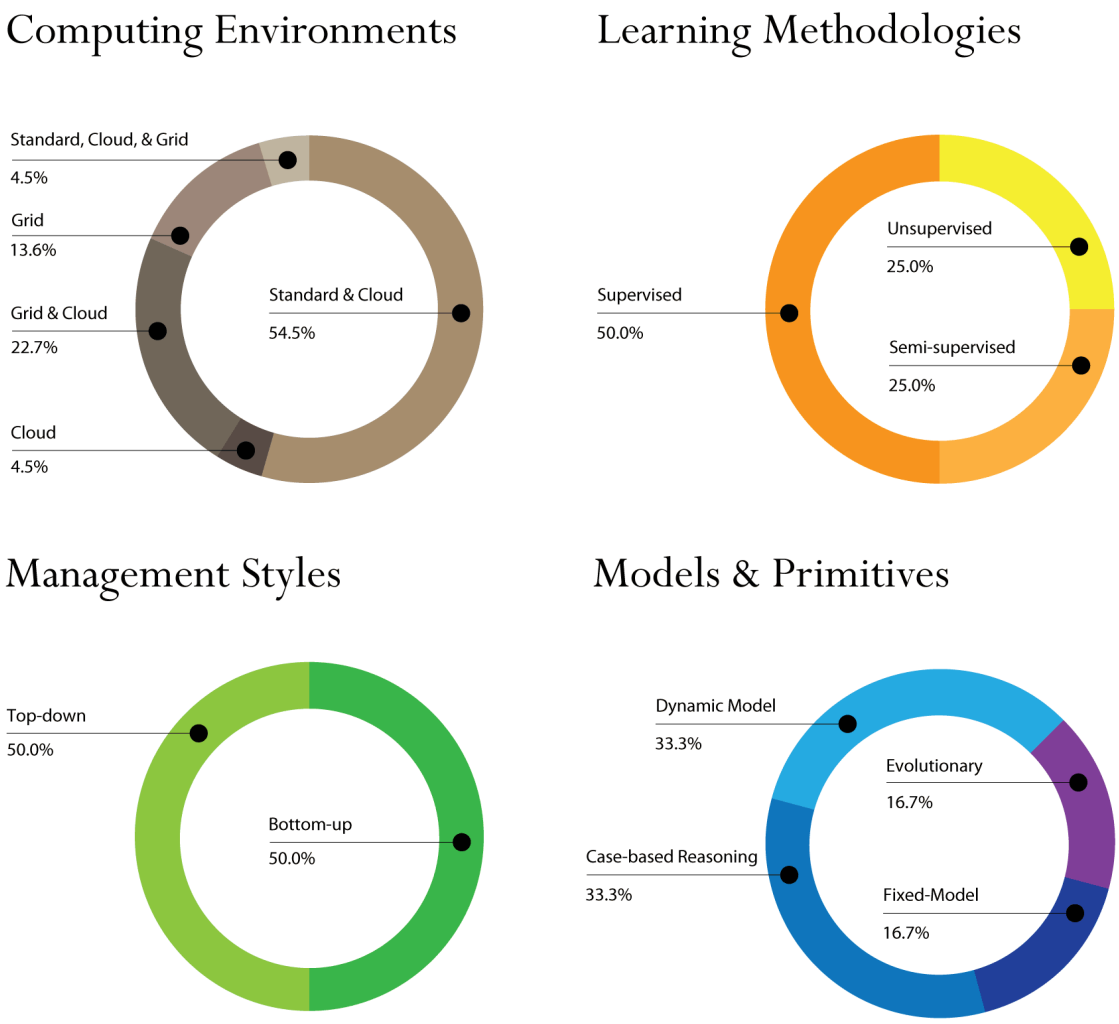

Figure 3.5: Relative Coverage of Different Self-Healing Techniques.

Evidence of specialisation in self-healing strategies is becoming increasingly more common as frameworks exhibit hybrid approaches for mobile [34] and centralised computing environments [37]. These approaches place a specific emphasis on leveraging different self-healing strategies based on the environmental suitability of the approach at run-time, and by anticipating resource availability. Notably, resource prediction is being leveraged more often where assumptions cannot consistently be made about the state of computing environment – particularly where resources are transient [39, 10, 33, 34], or virtualised [28, 92]. In these situations self-healing frameworks leverage multiple concurrent strategies to address greater degrees of systems volatility. Likewise, frameworks have leveraged various approaches for identifying and mitigating faults based on local and remote observations within their respective environments [72, 67, 93]. Figure 3.5 outlines the divisions of these approaches, visually.

Although the approaches used by self-healing systems are varied, there are trends as to which methodologies are being leveraged and under what circumstances. Systems within environments that exhibit a high degree of churn are more likely to leverage *ad hoc* management styles [34, 33, 10, 32], and learning methodologies that require less supervision [87, 28, 27, 37]. Conversely, frameworks that do not have stable systems membership are more likely to utilise a centralised form of systems management [70, 69, 87, 73, 67, 65], and exhibit supervised or semi-supervised learning methodologies [71, 72, 73, 34, 38, 67, 24, 70, 35, 66]. The predictability of a self-healing framework's actions are crucial in identifying operational requirements (*i.e.* SLAs), and are a defining factor in what behaviours are allowed or desirable in its respective computing environment. As behaviours are nearly solely defined by learning methodologies, is it clear that the relationship between management style and environment is linked with the degree of supervision required for its continual operation.

Using an *ad hoc* management style allows self-healing frameworks to leverage more autonomous strategies and learning methodologies. However, systems that engage in self-elected behaviours – particularly those that have not been previously vetted – have been shown to be inherently more risky when attempting to meet operational goals and less likely to produce reusable solutions [86, 84, 85]. It is for this reason that the use of centralised management techniques remains the preferred approach when environments are expected to exhibit a low rate of churn – the most notable examples being CBR and CBR-like learning methodologies [73, 68, 69, 65, 24, 67].

The advantages of self-healing approaches are directly related to their supervisory requirements. Although supervised learning methodologies have shown advances towards reducing human overhead, when compared to unsupervised methodologies, they have ultimately produced palliative results – particularly when executing recovery strategies. This is primarily due to

the fact that supervised techniques can only reactively detect faults [28], and that the solutions they generate often must be vetted via human intervention before being implemented. These solutions can become increasingly more complex to manage as the interdependency of features must be accounted for in subsequent self-healing strategies [31]. Such solutions are difficult to vet as often relationships between features are not immediately accessible either algorithmically or intuitively.

Semi-supervised and unsupervised approaches have shown stronger capabilities in ascertaining the root cause of a given fault, and producing non-palliative recovery solutions. In particular, the use of evolutionary programming techniques has demonstrated the unique ability to autonomously generate new systems configurations at run-time to mitigate faults [27], and the use of ANNs have been shown to correlate specific systems configurations with operational fitness levels to produce predictive fault detection [28]. These approaches show greater capabilities for autonomously self-healing of faults, but, like supervised methodologies, also come with certain restrictions.

Notably, the resources needed by unsupervised approaches can be much greater than supervised approaches, and frameworks leveraging these methods are not assured of finding a solution [28, 27]. These are properties inherent to the nature of search-space methodologies – either a predefined constraint is exhausted (*e.g.* time), or an acceptable solution is converged upon [80]. Exploration into these issues remains a separate field of study and outwith the scope of this survey – however it is clear they are deeply related to the viability of self-healing solutions.

3.4 Synopsis

Unsupervised fault detection within self-healing systems is in a relative early stage of development. The use of evolutionary programming techniques to successfully synthesise new, valid self-healing strategies have been present as far back as 2009 [26]. However, recent studies have emphasised the use of stochastic primitives to achieve this same goal – albeit with limited success [19, 18, 7, 28]. In order to quickly understand where common effort is being placed, an overview of these studies is provided. This includes their implemented primitive types, learning algorithm(s), and software suites – which in turn detail how information is gathered, and what faults are injected. Dividing these areas into distinct units for evaluation establishes the groundwork to describe commonalities in their respective implementations and helps in contextually understanding their results.

To advance the state of the art in self-healing systems research, a self-healing system must have an established baseline from which to understand its results, operate accurately using unsupervised learning to determine the root cause of faults, and use non-simulated, unlabelled, and contextually valid information to infer behavioural information. Successfully achieving these results requires a number of assumptions to be made, as some of their criteria are outwith the scope of this thesis – see Section 4.5.

In IBM's Autonomic Maturity Model the most advanced self-healing actions occur when general policies are applied to the system, and an *Autonomic Manager* self-elects its own behaviours. This idea represents a simplification of systems management into clear and concise goals, and it is for this reason that it should be emulated. Systems should be managed using a series of high-level operating goals, rather than being individually maintained at a technical level.

Evolutionary Programming offers some immensely useful extensions to computing behaviours that can, theoretically, address some of the problems self-healing systems research faces. The previously cited example of GAs being able to reactively synthesise new, valid configurations upon detection of a fault is one such approach. These techniques are unfortunately expensive, and time-consuming. In a professional environment, both of these resources must be minimised in terms of use. However, some of these techniques are more useful – and more importantly less costly – than others.

Fitness tests are primitives in Evolutionary Programming for evaluating the validity and effectiveness of a sample within a collection of potential solutions. These are akin to the performance tests this instance: Performance tests operate by examining broad, high-level criteria captured in SLOs. When used in self-healing systems, the overall behaviour of a system can be used to categorise its state. Additionally, by not identifying specific features to test, the application is left to determine the source of the fault when using performance tests because it only looks at the high-level behaviours. This allows for basic abductive reasoning to occur built exclusively from the application's logic.

Emulating the use of policies' high-level nature makes performance tests ideal for achieving the combined goals in existing research to use policy-based strategies to manage systems [94, 1, 3, 6], and roughly mirrors standard practice in existing computing environments where Operational Readiness Testing or SLAs are required. It also establishes the groundwork for feeding in the results of this experiment with existing evolutionary techniques – synthesising fault mitigation strategies using GAs being one such example.

Observing historical behavioural information is a tried and true method for learning and

predicting new information, however it is not an approach that has been widely adopted in self-healing systems research [7, 22, 8]. In fact, the majority of works still focus on supervised techniques that only take into account external learning from human-subjects on unexpected faults. Incorporating historical observation into windowed data-sets addresses contextual information problems. By understanding behaviours within a specific time period, predictions can infer the correct contextual information.

Which learning algorithm operates best under what circumstances is unknown. This is partially due to the fact that as a community their use isn't entirely understood (*e.g.* CDL) [95]. It is clear however that some algorithms are preferred as they are more conducive to the primitive being implemented, and to what feature sets they exhibit. For example Viterbi [25] and CDL [13] are both capable of multi-step ahead prediction, whilst Baum-Welch [96, 97, 98], Back-Prop [62], and Naïve Bayes [50] are not. As mentioned, these algorithms often come paired with a primitive – Viterbi operates on HMMs, and RBMs leverage CDL, respectively – but it is not always the case. Fully recurrent networks (*i.e.* unrestricted) stochastic primitives remain a grey area, particularly as most do not have practical learning algorithms with the potential exception of Generative Stochastic Networks (GSNs) (see Chapter 6, Future Research).

Understanding and forecasting behaviours within a system are not enough to make an assessment of the effectiveness of self-healing systems methodologies – either in terms of cost or complexity. To do so would require a comparison against human-subjects of which there is currently no known public research. Similarly, direct comparisons between self-healing systems appear to be largely unavailable. To understand the strengths and weaknesses of these studies, they must be compared.

The following chapters detail the experiments and results provided in this thesis. They focuses on expanding the discussed approaches, analysing the validity of historical feature behaviour, and outlining a new, accurate approach for determining both the presence and root cause of faults.

Specifically, it emphasises using feature change data to address the problems mentioned in the previous section by demonstrating the effectiveness of several types of stochastic primitives using unsupervised learning to autonomous identify the root cause of faults. This approach demonstrates not only the effectiveness of different types of stochastic primitives, but also accurately identifies faults using both abnormal application termination and human-initiated faults (DFIs and ACCs, respectively), mitigates convergence by using a windowed dataset, compares greedy and lazy approaches, uses non-simulated data, and shows how noise can be an indicator of expected feature behaviours.

Positive results from these experiments demonstrate possibilities for reducing the resource requirements associated with autonomous fault detection, and autonomously discovering relationships between features. This includes understanding the benefits of mitigating convergence through the use of a windowed dataset, the potential of narrowing the search-space when using GAs to synthesise valid systems configurations, and using CDL to forecast multi-step ahead feature behaviours.

CHAPTER FOUR

AN AUTOMATED APPROACH FOR IDENTIFYING FAULTS

This chapter briefly summarises the problem statement of this thesis before theorising and then postulating on a new approach to automating fault detection using stochastic primitives and unsupervised learning. Afterwards, research questions, contributions, and the experimental design for testing this hypothesis are detailed using two separate but related approaches: One experiment uses HMMs and ANNs, whilst the other uses RBMs. It then describes the limitations and threats to validity posed to these experiments, followed by their respective implementations.

This chapter concludes with an outline of a comparison against a similar experiment leveraging SOMs before leading into the following chapter, where results and a discussion are provided.

4.1 Problem Description

The question remains: How can we further automate the behaviours of self-healing systems whilst reducing the operating costs of large-scale computing environments? Accurately identifying the root cause of a fault should allow for less human oversight and reduced costs. However, identification of the root cause of faults is non-trivial. A number of problems exist between the detection and subsequent correct identification of the root cause of a fault including

ambiguity in information, contextual inference, etc. A small step forward would be to automate the diagnostic process that human engineers perform by short listing potential avenues for investigation.

It is possible to leverage feature changes of a faulting system to accurately automate the identification of their root cause(s) if, prior to the fault, working configurations have been observed. This approach works for either singular VMs or physical computing systems.

It assumes that the features associated with (*i.e.* leading to) the root cause have been observed in both several working configurations and at least one faulty configuration, that there are measurable metrics for deciding when a system is in a normal operating state (*e.g.* SLOs), and that there exists an uniquely identifiable, feature observation or collection interface – such as the */proc* filesystem or Windows Management Instrumentation (WMI).

There are two classes of faults this approach attempts to correctly identify the root cause of: DFIs, and ACCs. The former deals with faults directly caused by logic failures – such as a crashed stack, or abnormal application termination – and the latter focuses on user-related errors – such as switching off a service at the wrong time. Both of these types of faults deal with application failures, or more specifically "services". It does not matter where or how these faults are generated, so long as they have an observable feature and that feature is observed by the FDF.

This approach is expected to work on other operating systems besides Windows regardless of the primitive used, as is indicated by the external comparison [28]. However, no comparison of the FDFs are made on other operating systems due to the proprietary nature of the C# language and WMI.

4.2 Approach

Determining the root cause of a fault is a hard problem – one that may help reduce costs in mitigating downtown in computing environments. The approach proposed in this thesis demonstrates novel capabilities for analysing of the root cause of faults within a computing system, with the aim of eventually producing demonstrable cost reduction capabilities. Uniquely, it shows that monitoring the content of the data is not strictly necessary to determine the root cause of a fault as monitoring the pattern of changes in the observed data can be sufficient. Using machine learning techniques this can be automated, and provide a helping hand to existing computer operating procedures.

This approach builds on prior art – chiefly from the Autonomic Computing initiative – but also leverages Machine Learning and Computational Intelligence techniques. The approach operates in two stages:

First, an application periodically samples feature behaviour data. This information is transduced into vectors which form the basis for future analysis and forecasting. Second, the data is labelled – a process that occurs through performance tests. If a system passes a number of high-level objective goals and policies, the data at large can be assumed to be in a 'good' state. If any of these tests fail, then the opposite is assumed and an analysis is performed against the likelihood of the expected and observed behaviours using trained stochastic primitives via the known 'good' feature data.

Finally, once trained, if any of the specified performance tests fail, then the primitives forecast feature behaviour to varying degrees – both inherent to their respective learning algorithm(s), and by how much training data is present. Any mismatches are detected and returned in a list ordered by descending likelihood indicating the potential root cause of the fault.

This thesis explores two possible implementations of this approach – one using a greedy data ingest mechanism (via ANNs, and HMMs, Figure 4.1), and one using a lazy data ingest mechanism RBMs (Figure 4.2) – both of which are detailed in Section 4.3 along with examples of their operation. In all other ways, aside from the learning modules, the approaches are identical.

To accurately identify faults, the FDFs require a user to provide:

1. A polling interval (in milliseconds),
2. A 'learning module', and
3. A set of performance tests.

A polling interval specifies how often the systems' feature data should be gathered, evaluated, and stored. The learning module consists of a stochastic primitive and an associated learning algorithm. Using the AForge.NET and Accord.NET frameworks, it is possible to select a number of previously built stochastic primitives and and associated learning algorithms. However, a user may also specify their own primitives and learning algorithms. Performance tests consist of user-designed code that evaluate the state of the system. This are simple tests written into the application to verify the health of the system and are akin to SLOs. All of the specified tests must pass during each polling interval.

A user may also update the maximum number of samples to retain. This allows for greater

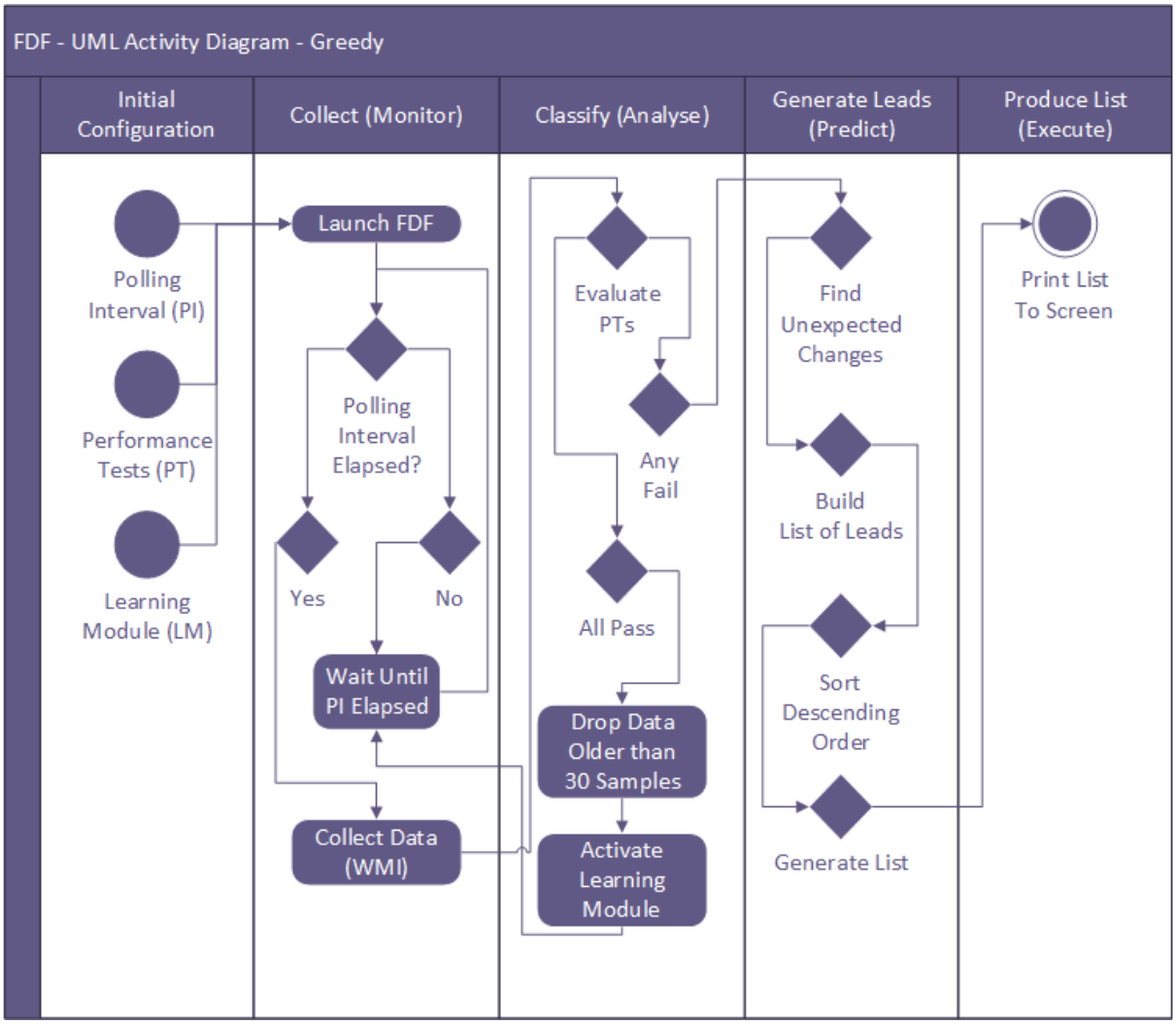

Figure 4.1: Fault Detection Framework Logic & Architecture Diagram using Greedy Ingest. The FDF leveraging ANNs and HMMs operates by updating its primitives as soon as feature data is recovered from the system.

control over the window of observed data desired to be retained by the FDFs. For example, a higher frequency collection rate of once per 5 seconds using the default value of 30 maximum samples would only allow for a 150-second window of observation. It is unlikely this window will be sufficient to capture changes in feature behaviour outside of this time-span. Increasing the maximum number of samples thus increases the window size, and by adjusting the maximum number of samples and the polling interval, it is therefore possible to control fidelity of the information as well.

The FDFs are configured by default to run in a Windows VM to test the performance and state of Internet Information Services (IIS), Microsoft's proprietary web service. No changes are

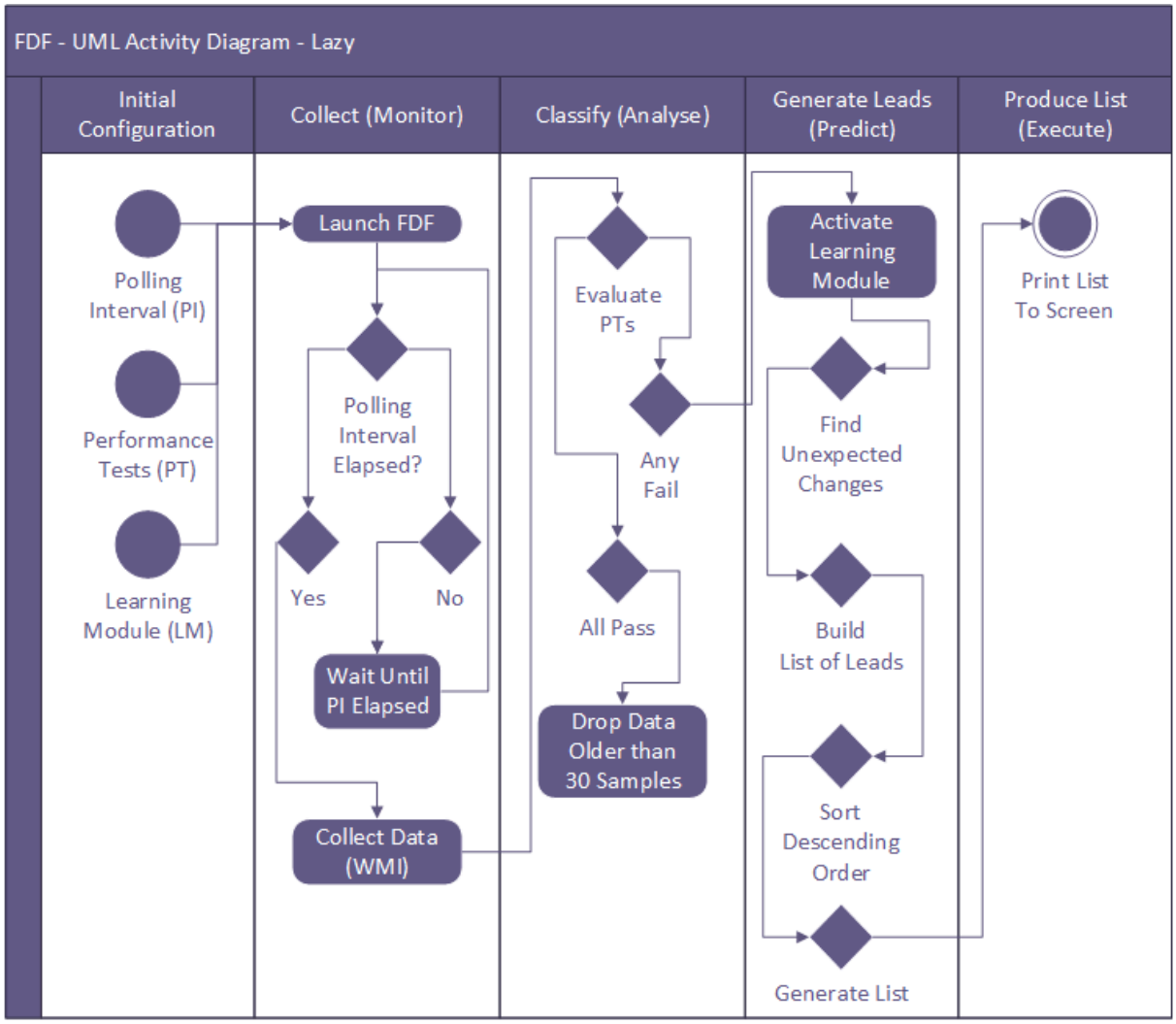

Figure 4.2: Fault Detection Framework Logic & Architecture Diagram using Lazy Ingest. The FDF leveraging RBMs operates identically to the FDF that uses ANNs and HMMs except with a lazy ingest mechanism for feature behaviour data. Primitives using a lazy ingest are only trained upon fault detection.

required to the source code to revalidate the experiments described in this thesis. Although it is believed that these results can be generalised to other operating systems, no attempt is made to demonstrate this.

4.2.1 Running Example

Step 1 (Optional) A user provides the FDFs with the three requirements described in 4.1, and then compiles the code.

Step 2 The application is run and then waits in the background whilst the computing system operates normally for a desired period of time – in the case of these experiments, 5 – 30 minutes. After each polling interval the framework prints to screen memory usage and the collective state of the performance tests as "System State" (Figure 4.3).

```
DataSet Collection: WIN-2HRNFJ1VLUO_0424_0953     369919    System State is: True
DataSet Collection: WIN-2HRNFJ1VLUO_0424_0954     739340    System State is: True
DataSet Collection: WIN-2HRNFJ1VLUO_0424_0955     1108760   System State is: True
DataSet Collection: WIN-2HRNFJ1VLUO_0424_0956     1478179   System State is: True
DataSet Collection: WIN-2HRNFJ1VLUO_0424_0957     1847602   System State is: True
DataSet Collection: WIN-2HRNFJ1VLUO_0424_0958     2217023   System State is: True
DataSet Collection: WIN-2HRNFJ1VLUO_0424_0959     2586443   System State is: True
```

Figure 4.3: Running Example – Successful Data Collection via FDFs. Cropped image showing successful data collection when running an RBM-based FDF.

Step 3 Upon injection or detection of a fault, the FDF should break the loop it is currently operating in and provide a list of fault hypotheses in descending order of likelihood (Figure 4.4). If the monitoring loop is not broken then a False Negative must be accounted for by hand.

Figure 4.4: Running Example – Fault Identification via FDFs. Cropped image showing a sample of an FDF result screen. The full sized list has been truncated to save space but can contain 5 to 300 leads.

4.3 Experiments

This section broadly describes the approach taken whilst the technical implementation details are described in Section 4.6. The implementation of this approach details what is necessary to run the framework and use it to identify faults. The rest of this section on approach describes the overall operating strategy of the FDFs.

Both FDFs periodically sample behavioural feature data from a local system using the WMI. This data is then converted into vectors for each individual observed feature of a specified window of time – which in turn provides contextual inference and avoids convergence problems. Vectors are used to train stochastic primitives for predicting the behaviour of features to analyse features for potentially errant behaviour.

Errant behaviour is determined by comparing the actual and forecasted changes for each monitored feature when an SLO fails. Any feature that does not exhibit the predicted behaviour by its respective stochastic primitive is short listed as a potential lead for the root cause of a fault. Prioritisation and ordering of these leads is provided by sorting the leads in the inverse likelihood of the change observed – less expected events are moved further toward the top of the list.

In order to determine if the data should be used to train a stochastic primitive, a series of performance tests determines the overall health of the system being observed. How these tests operate is described in Section 4.6 – but, broadly stated, a passing series of performance tests reinforces behaviours in the primitives through training, and a failure sends a signal to begin forecasting and temporally comparing feature behaviours.

Using this approach validates or invalidates the hypothesis by testing the forecasting capabilities of stochastic primitives. If the primitive successfully indicates the correct root cause of the fault after a performance tests fails, then it is clear that the forecasting abilities are working correctly. Conversely, if the primitives do not indicate the correct feature then the hypothesis is not supported.

Fidelity being a chief concern in these experiments, different volumes of data are used to show trends in the approach. Specifically, how much data is needed to train the primitives is explored through the different volumes of input by using 5, 10, 15, 20, 25, and 30 samples. Each sample corresponds to one minute intervals. These values were chosen arbitrarily with the intent to provide a reasonable enough time for the system to accommodate changes.

This approach operates on a few assumptions. The first is that without changes within the observed features' values the stochastic primitives used in this experiment would not function at all. This is one of the reasons for waiting 60 seconds between samples. The second is that the fault must lie within the observed features' behavioural data to have a chance of being accurately indicated. One test explores beyond this assumption with surprisingly positive results, but those results are, expectedly, not accurate.

Several key aspects are addressed with the FDFs that appear to be missing from current self-healing systems research. In addition to open questions about fidelity and a lack of basic comparison of performance between different types of primitives, little research exists between studies that explore solutions in the aforementioned fashion. Specifically, the simple observation of feature changes rather than their explicit values had not been examined. Additionally, only one study so far has attempted to use evolutionary programming techniques to explore recovery strategies [27]. This work attempts to address some of the search-space challenges

within that study by providing a mechanism for guiding GAs (see Chapter 6, Future Work).

4.3.1 Hidden Markov Models & Artificial Neural Networks

The first experiment leverages HMMs and ANNs, to periodically sample configuration data via an interface, then classify this data using a series of performance tests. Based on the collective results of these tests, the information is categorised as either being in a *good* or *faulty* state. Afterwards, the FDF takes one of two potential actions.

The first action is to update a local data-store. When the system passes all of its performance tests, the existing data-store is examined to make sure its total number of configuration samples does not exceed the maximum threshold. Any data that is beyond the maximum number of data-sets is dropped. The primitives are then greedily trained using their respective learning algorithms using the remaining and latest configuration samples.

The second is to perform an analysis on the system's feature behaviour data. Features that show changes between the previous 'good' sample are compared to the 'faulty' configuration. If a change is noted, it is short-listed for comparison. This is an optimisation technique that reduces the maximum number of features for investigation. Any changes are fed into their respective stochastic primitive where the likelihood of the change is then compared to a forecasted value.

The differences between the expected (*i.e.* forecasted) value and the actual value (located within the *faulty* configuration) provide a measure of confidence or likelihood for the potential cause of the fault. Once this comparison is complete the feature is added to a list of potential root causes. Finally, this list is sorted by highest likelihood starting with the first (0^{th}) index (Figure 4.5).

Using this list, metrics are generated via the FDFs that indicate precision, accuracy, prediction time, the aforementioned confidence value, and the total number of leads generated. The conditions of these metrics, such as what constitutes True and False Positives or their respective Negatives are explained in Section 4.3.

Testing is done through fault injections. These take two forms: DFIs and ACCs. The details of their implementations and differences are discussed in Section 4.3, but the theory behind these two approaches can be summarised as examining the differences between software errors and human errors, respectively. In each case the root cause is known to the administrator but not to the FDF. This allows for validation of the result provided by the FDF, and an unbiased attempt at identifying its respective source – the latter being the primary goal of these experiments.

Both the ANN and HMM approaches operate using single-step prediction that has been

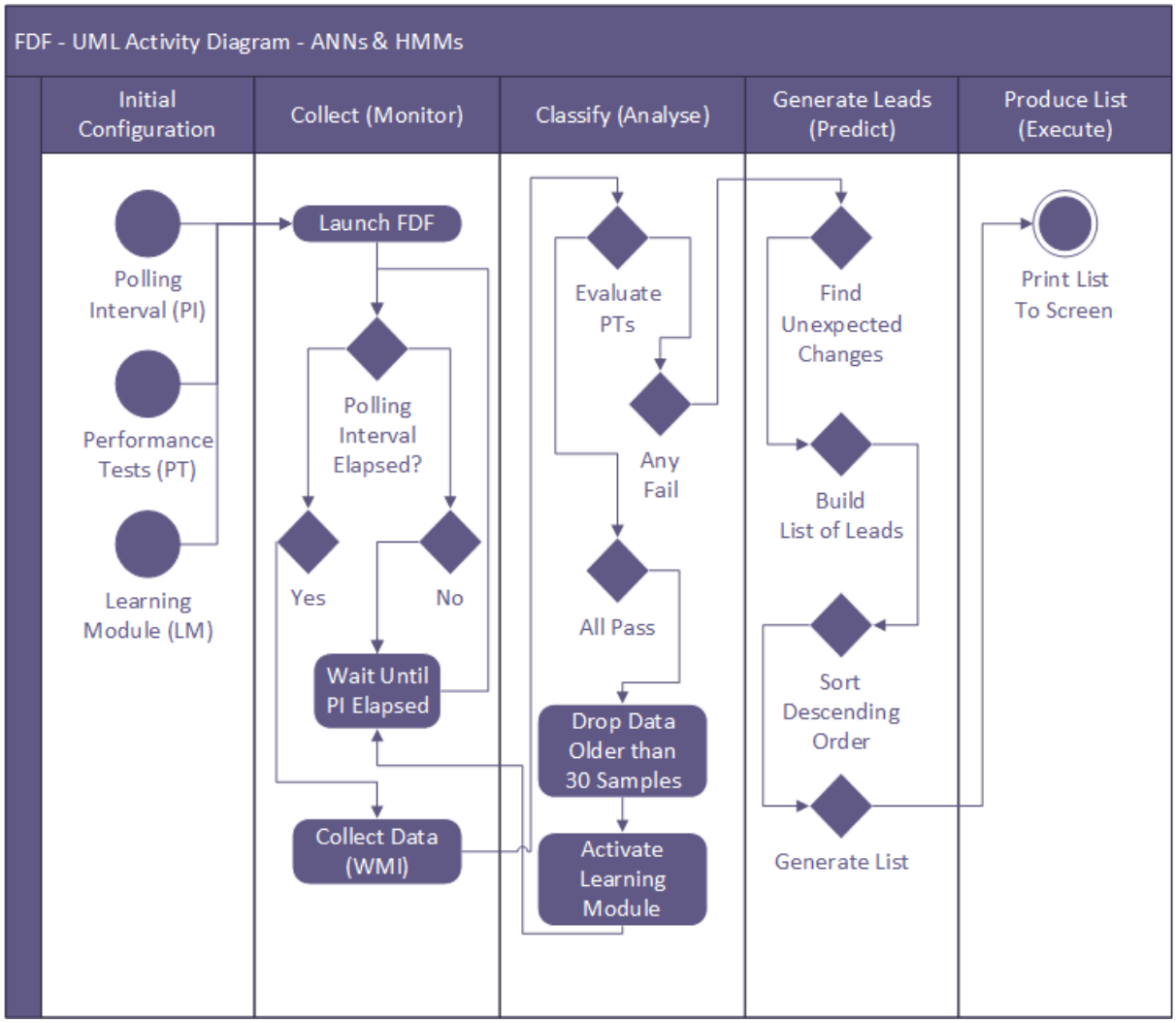

Figure 4.5: Fault Detection Framework Logic & Architecture using Hidden Markov Models and Artificial Neural Networks. Fault Detection Frameworks are provided three inputs, set to run, and then injected with faults at varying time intervals. The result is an ordered list of leads based on forecasted feature behaviours.

implemented in a reactive manner. Whilst unsupervised learning is generally meant to forecast behaviours into the future, this experiment is meant to be a baseline to determine the accuracy of future endeavours. Understanding their operational capacities is therefore emphasised.

Lastly, each of the FDFs requires basic instantiation before operating. As mentioned previously this book does not centre on self-configuring (*i.e.* self-provisioning) methods and an initial, minimal setup is required. The FDF must be provided with a polling interval, a set of performance tests from which to ascertain the system's overall health, and a stochastic primitive with a coupled learning algorithm.

4.3.2 Restricted Boltzmann Machines

FDFs that leverage RBMs operate under nearly identical assumptions and conditions as those used for ANNs and HMMs. The polling interval, performance tests, and learning modules are provided to the FDF. Afterwards, it is allowed to run for 30 minutes before being subjected to the same DFIs and ACCs.

The primary difference when using RBMs is how the primitives are trained. In the former experiment a greedy approach is used in conjunction with a windowed data-set. This necessitates that the primitives are destroyed and retrained after every successful data collection. Thus, although faster predictions can be made under these conditions, it also requires more persistent use of a system's resources. This can create an artificial limitation if the number of features being parsed grows to a size greater than the system can parse within the polling interval.

To alleviate this problem the RBM approach uses a lazy implementation. Data is not directly parsed until a potential fault has been detected. The caveat to this is that the FDF is unable to return potential root causes as quickly as its counterparts.

A secondary difference exists in that some of the vectors used to train the RBMs are partially incomplete – this is intentional. Although the goals of these experiments remain intact by not using simulated feature behaviour data, the requirements of the RBM in how it is trained requires some unique properties. Specifically, the RBMs cannot be trained without a complete dataset because the learning inputs must be vectors of equal size.

There are two ways to address this problem. The first is to wait for double the amount of time for a maximum window containing the maximum sample-size number of configurations to populate – in this case 60 minutes. The second is to assume vectors the size of the data-set window (*i.e.* maximum sample size) and populate them as more information becomes available.

In the former, the amount of data being used by the RBM is greater than those in the other experiment. As the total time to observe the system is a key variable in understanding how quickly the primitives can be trained, this is ruled out as a potential option. This additionally allows the experiments to use the same amount of time to attempt to generate results.

RBMs are trained using half as much data as they could otherwise use. The vectors are instantiated based on the provided maximum sample size. Each vector contains a series of values, with each value indicating a certain observation of feature behaviours – *change*, *no change*, and *unknown* – (1, 0 and *null*), respectively. Using the latter of these indicators, vector information is gradually changed from *unknown* to their correct, respective indicators.

In contrast, the HMM and ANN FDFs continue to increase their vectors by adding one additional index until the maximum window size is reached; a third value is unnecessary. However, in this case it was important to try to distinguish between all three states for the sake of accurately inferring the vectors' information. This is further described in Section 4.5.

4.4 Limitations

Feature prediction has proven to be a viable method for determining the root cause and potential for impending faults [28, 20, 19, 18]. However, the accuracy of these approaches does not appear to be compared outside of this thesis. A baseline for this information is presented and analysed in order to understand relative effectiveness and performance criteria between studies but remains to be validated (Chapters 5 and 6, respectively).

Related, as more information is used to train stochastic primitives, their effectiveness diminishes due to convergence. To mitigate convergence, stochastic primitives need to have a time-relative, windowed dataset from which to infer behaviours. This means using the more computationally expensive route of expiring old data and retraining primitives as needed. It is worth noting that time to infer and predict failures is also important. Without a speedy prediction, forecasting a failure may not occur within enough time. This means balancing greedy and lazy approaches and average response times, with their respective resource utilisation. Some of these topics are superficially addressed, but not specifically within the context of self-healing systems.

Contextual inference when understanding feature behaviours is important but assumed to be difficult to attain in computing environments. To partially address this problem, the use of windowed data-sets is required, *a priori*. Using time-specific sets of information for analysis avoids problems with convergence and over-training primitives – a topic which is discussed in several existing experiments [99, 28, 88, 100].

Stochastic primitives leveraging unsupervised learning currently represent the best known approach for multi-step ahead forecasting of feature behaviours [7, 1]. The relationships between features are too complex to be modeled accurately in real-time by any human due to the dynamic and rapid nature of their interactions. However, as as many faults are suspect to have a number of dependencies in their root cause identification, performing such an evaluation appears to be necessary for fully featured self-healing systems frameworks.

Accurately labelling data in an autonomous fashion is not a problem this research claims to solve, nor are recommendations provided for improving learning methodologies. Instead, these

are problems that are resolved through operational policies and direct comparisons, respectively.

Each feature has its own dedicated stochastic primitive. Although costly, the nature of broad, context-free data collection contrasted with minute predictions at the feature level make this requirement necessary.

The nature of the general approach described in this thesis in Section 4.2 necessitates certain questions – such as "Should the primitives be trained every time data is collected, or only when a potential fault is present?', and "What data should be used to avoid the pitfalls of emulation?" Respectively, both lazy and greedy implementations are investigated along with controlled, direct injection of faults in specific threads along with adverse configuration changes to provide realistic user behaviour data within a system's feature-set(s).

Some of the data in the RBMs vectors are incomplete. It is not possible to completely fill the vectors within the time allotted with a full value set – to do so would require 30 minutes of waiting, followed by 30 minutes of training. Instead, the vectors are populated with values indicating a lack of observed knowledge.

Specifically, RBMs require data to be presented in a square matrix of values. In this case that matrix is bounded by the window size as defined at compile time. However, this isn't compatible with the default instantiation of the FDFs and the iterative tests using the aforementioned subset sizes. To address this, 'no data' values were implemented into the RBM inputs so that the input data sizes were the same across all iterations between both frameworks. This allowed the RBM to recieve, effectively, 'no-op' controls in the learning algorithm. The alternative was to allow the RBM double the time to acquire data which would then make the tests unequal.

By approaching the experiments in this fashion, the RBMs use the same amount of data as presented in the other experiments. Even with this limitation – imposed for fairness between comparisons – the results are positive for this approach (Chapter 5). However, the potential to perform better, overall, readily exists. If fully trained, RBMs appear more likely to continue to demonstrate more accurate results.

WMI is ineffective at gathering feature data in a manner that is uniquely identifiable. Although touted as a mechanism for do exactly this, the notion of such structure in the data it returns is entirely absent. In addition to lacking a unique identifier for the information it provides, other problems include the nature of the data gathered by WMI itself.

Deeper investigation showed WMI is not a stand-alone service, but a conglomerate of registry values, COM+ interfaces, and the occasional hard-coded performance counter. Whilst accessing this information in one place is more accessible than gathering it separately, there is limited

reliably as to how often the gathered information is updated. This appears to be due to the individual policies of each of the aforementioned conglomerate pieces of code and their individual update policies not aligning.

A lack of reliability in this respect is a problem when trying to document periodic feature changes. To overcome this, each class within the WMI service had to be explored individually and tested before use.

The described properties of WMI are not documented widely and – with the exception of the unique identifier – were only noticed after extensive testing to ensure validity in results. In order to ensure consistent results that were up to standard for both academic and industry rigour, these problems had to be addressed. This is one aspect to how and why the initial tests were created, including the decision to use one minute intervals.

If a fault is caused through feature locality – the notion that a fault manifests only when two or more features have specific properties at the same time – then it is much harder to diagnose using this approach unless they occur at the same time. This is a problem that was encountered three quarters of the way through the research that would have been interesting to attempt to address in the experiments. Instead, they have been anticipated by preparing for and accommodating learning algorithms that can use a single output to generate a series of probabilistic inputs after an initial learning phase (see Section 6.3, Future Work).

Details of the implementations of these approaches are discussed further in the remainder of this chapter. In summary, the application of this theory has met with overall success (Chapter 5), but many questions remain to be answered (Chapter 6) – such as which algorithm is best under which circumstances? To help address these questions – and because results are best when compared with impartial, external data – a direct comparison with a similar study is included (Sections 4.7.2, and 5) [20].

4.5 Threats to Validity

4.5.1 Construct

Confidence is intended to provide a human-readable metric for the expected margin of error used by the learning algorithm. Overall this metric works as intended, however because Baum-Welch uses a proportional probability, direct comparisons between other machine learning approaches is not directly possible. This can cause some confusion if interpreting

the results as “100%” likely, despite misdiagnosis.

Time Taken measures the amount of time taken between when a fault is first suspected and when the list of descending ordered fault hypotheses is generated was done with great care. This was done to the point of using a programming object (*ElapsedTicks*) that should produce similar results on any system, despite even minute differences in CPU clock cycles and frequencies (see definition *ElapsedTicks*). However, the differences between lazy and greedy approaches require an extra layer of inference to perform direct comparison in some capacities due to a shift in where the bulk of the computation occurs. This means using averaged results to provide the most accurate outputs, which is what is provided. These value are converted to milliseconds (ms) for comparison and easier consumption.

Total Leads are intended to represent the total number of fault hypotheses such that an understanding of the overall work-load of the FDFs can be easily ascertained. In this capacity, the Total Leads metric operates as intended and helps correlate expected increases in time taken and changes in fault position.

Fault Position is the primary metric upon which the results of these experiments are based. An ideal result is to have an expected feature reach position 0, along with other associated features directly below it, if appropriate. It is believed that it serves this capacity accurately, including the outcome that some high entropy variables were predicted to be difficult to remove from the lowest (0^{th}) position.

4.5.2 Internal

Polling Interval is a value that determines how often the computing system should be sampled for feature data. Increasing this value to a point that exceeds the capabilities of the computing system to return data will result in failures at the application level and cause the experiment to produce unusable results. Additionally, sampling information too infrequently could mean losing fidelity or understanding of when feature changes are occurring. A user must elect a Polling Interval that is appropriate for their situation – a definition that is intentionally left open to interpretation so that a variety of services and circumstances can be accommodated – and will provide, when combined with the maximum number of configuration samples, an appropriate window of inference.

Maximum Total Configuration Samples is related to the Polling Interval and helps determine the total window-size for making inferences and for forecasting. Increasing this value al-

lows for greater fidelity or larger windows of inference, but it comes with higher memory utilisation and longer computation times when producing a list of fault hypotheses.

Epoch Count directly impacts the forecasting accuracy of primitives such as RBMs. The value used in these experiments was briefly explored by hand but ultimately the default value of 5,000 was chosen after some anecdotal exploration of expected cycle times in other implementations of RBMs. Changing this value could deeply impact results, particularly variance.

Faults (All) represent the core of the experiment. Injecting a fault that does not have an associated, observed feature is assumed to be unlikely to produce accurate results. In one instance this was tested and a partially accurate analysis did occur when using the maximum number of samples. Specifically, unplugging an upstream router – which the FDF could not have known about or monitored – produced an arguably correct diagnosis of a change in network throughput. This is considered to be an unusual result and similar tests are expected to fail. Additionally, the type of fault injected matters in relation to the accuracy of the output. This is discussed further in the Results Chapter (Chapter 5).

4.5.3 External

Noise is an inherent problem in understanding feature behaviours; it may not be possible to address it in all circumstances. Certain features or properties – such as free disk space and the total number of active threads, respectively – can be extremely difficult to predict. This creates problems in understanding if these traits are associated with a fault using heuristic learning in a number of ways – particularly when using random values used to instantiate stochastic primitives.

The approach described in this thesis attempts to accept that some features should exhibit a certain level of randomness in their behaviours and then be alerted on if they start displaying predictable patterns. This may impact reproducibility for specific tests, but overall results should remain larger familiar in terms of overall output.

WMI has a number of limitations, and no assurances can be given outside of systems that are configured differently – including the version of software present on the system, e.g. WMI. Additionally, which WMI classes are selected for observation will impact results. Details on the specific classes used for observations of the features observed in these experiments are provided elsewhere in this chapter (*e.g.* Figure 4.1).

Drift – the differences in configuration states between experimental trials is referred to in this thesis – will impact results, even between similarly configured systems. This is based on the assumption that that computing systems undergo continuous changes. As a consequence evaluating exactly identical conditions between systems is not a state that is easy to achieve. A best effort is made to reduce drift as much as possible by running multiple tests via the same saved VM state. However, it is acknowledged that drift can have an impact on results and that the extent of that impact isn't entirely understood.

4.6 Implementation

Implementation of all FDFs is done via a small (~40 kilobytes), console-based application written in C#. This application uses the WMI framework to periodically collect data based on a polling interval. Information collected in this manner is either immediately analysed upon detection of a fault, or added to a collection of datasets for later use. Primitives are built and trained through the Aforge.NET [101] and Accord.NET [102] frameworks.

There are other mechanisms for gathering feature behaviour information from a system, however this approach was initially thought to be the least difficult. WMI allows for a wide range of feature data to be pulled. Existing methods within C#, and the .NET framework provide an established interface for polling information. C#, WMI, and the combined, existing Machine Learning frameworks, minimise development requirements; a property that remains absent from other operating systems.

The polling interval is provided at compile time and determines when the WMI service should be queried. When combined with a maximum number of datasets to keep in memory, an elastic measure of control is afforded to testing conditions. Furthermore, by specifying the maximum number of configurations to keep in memory, a window of inference is created. In this case, assuming a 300 second interval, and a maximum of 50 samples, a 250 minute window would be created for contextual learning. There are expected benefits and disadvantages to shorter versus longer windows – such as stability in predictions, and time to fully train the framework(s), respectively. The adjustment of these properties is not explored in favour of steady results. Although both values are fully adjustable, the experiments in this book use a polling interval set at 60 seconds, and the total size of the dataset collection is limited to 30 samples.

Data collected from the WMI interface is stored in a dataset. Each dataset is referenced within a list that corresponds to its collection time, and contains a collection of tables that individually correspond to their respective WMI class. Each table contains a series of values – some of which

are unique, and some of which are not – this is an artefact of Microsoft's implementation. For example, if the value for a system's fully qualified domain-name is present in one WMI class, it may also be present in another such class.

WMI Class Name	Unique Column Identifier
Win32_BIOS	► Version
Win32_ComputerSystem	► Caption
Win32_DiskDrive	► DeviceID
Win32_LogicalDisk	► DeviceID
Win32_NetworkAdapter	► Caption
Win32_OperatingSystems	► SerialNumber
Win32_PhysicalMemory	► BankLabel
Win32_Processor	► ProcessorId
Win32_QuickFixEngineering	► HotFixID
Win32_Service	► Caption
Win32_SystemAccount	► SID

Table 4.1: WMI Classes – Names & Unique Column Identifiers. This table illustrates the classes and the columns used to uniquely identify rows within the sampled WMI data.

The WMI classes are hand-picked in these experiments based on whether or not they might be useful to an engineer diagnosing faults with a computing system running a web-service; all of their available data is sampled (Table 4.1). Some of the information is expected to be more directly applicable – such as Win32_Service – whilst others to supply supporting information (*e.g.* Win32_Processor).

Unique identifers for these classes have all been selected for the same reason: Their values are unique and expected to remain unchanged. This takes the place of a unique identifier being supplied natively by WMI. Although it was the original intention of the experiment to gather and compare datasets using the previously mentioned criteria using only WMI in its 'out of the box' state, critical limitations were discovered in Microsoft's implementation that necessitated a few fixes. Fixes include the use of bridging code between the organisational structures of the FDFs and WMI under the following scenarios:

1. As mentioned the WMI framework does not provide a unique identifier for any of the data it returns. This means comparisons based on unique values become nearly impossible

without a linear search through every tuple returned within a single class. As a partial fix, a dictionary of WMI classes is provided to the FDFs that indicates a column in which a value can be used to uniquely identify the tuple. This is a mechanism that had to allow for comparisons between the same features' values – a necessary capability for analysing feature changes.

2. The doubling of data is common in WMI. In many places the same information will be returned, but this will be viewed as a different or nominal piece of information. Because there is slight drift between when the system queries the classes, sometimes these values can appear to change – this can lead to false positives, or misrepresentative data. For the most part these values are ignored and it is left to the primitives to properly diagnose these features on their own. The end result is that memory is not used as efficiently as it could be.

When the WMI framework is queried, the respective WMI tables and datasets are populated. In total, about 7,500 features are extracted every 60 seconds. At the end of each collection, a categorisation process occurs through the use of performance tests. Performance tests validate the responsibilities of the machine running the FDFs. In this case the machine's primary purpose is to act as a web-server for both internal and external clients. As mentioned in Chapter 4.3, Performance tests verify a series of high-level processes and functions and emphasise a policy-based approach to systems administration. These tests are implemented as described in Table 4.2.

Once a dataset is categorised as either valid or invalid the application will either update its predictive capabilities or it will look for anomalies, respectively. The dataset is determined to be valid if it passes all of its performance tests. If this occurs, each property within the collection of datasets is evaluated against itself. The hardest part of this procedure is uniquely identifying the objects that have been queried – a feature that is surprisingly not natively supported in WMI,

The FDFs in this experiment leverage one of three learning algorithms. The FDFs that utilise an HMM leverage the Baum-Welch algorithm [98, 97, 96]. This algorithm has been chosen due to its suitability with HMMs inherent forward–backward learning, and its ease of implementation via the aforementioned AForge.NET [101] and Accord.NET Frameworks [102]. Conversely, the FDFs that leverage ANNs utilise a Naïve Bayes approach. Each learning algorithm is responsible for processing observed feature behaviours into probabilities, which are used in conjunction with the FDF's classification of the datasets collected via WMI.

If the dataset is determined to be invalid, the feature's behaviours are analysed by the FDFs for unexpected changes. Any property that does not match the FDF's predicted values is added to a list of potential faults, along with a confidence value. As long as the fault source is collected

Test Name	Description
Physical Disk Access	► Writes to physical disks. Fails on any system-level exceptions including permissions, free space, and others.
DNS Service Availability	► Resolves three separate fully-qualified domain names: google.com, yahoo.com, and microsoft.com. Fails if all three sites fail to resolve.
WMI Accessibility & Physical Memory	► Queries the WMI service to ensure the total free physical memory is > 0 bytes. Fails on any state where the WMI service cannot be queried, or memory is = 0 bytes.
ICMP Ping Test & Internet Connectivity	► Performs ICMP Ping tests to localhost, and **K**, the root DNS server at 4.2.2.4. Fails on any state that does not return a successful reply to both sites.
IIS Service State (W3SVC)	► Uses a Windows Service Controller object to determine the state of the W3SVC service as either *Running*, *Stopped*, *Paused*, *StopPending*, or *StartPending* states. Fails on any state that is not *Running*.
HTTP Request (localhost)	► Performs an end to end HTTP query to the localhost. Fails on inability to connect and complete a request to the service – all errors are considered valid if passed from the web-service, *e.g.* 400, 404, 500.

Table 4.2: Performance Tests – Names & Descriptions.

within the WMI data, the potential exists for the root cause to be provided. Determining what constitutes is sufficient potential is one of the goals of this experiment.

Implementation of the FDFs occurs using three separate but nearly identically configured VMs. Each VM runs Windows 7, IIS 7.5, and one of three versions of the FDF. The VMs are clones from a single initial image which consists of an identical base configuration and hardware specification: one 3.4 GHz Intel i7 4770 CPU, with one gigabyte of RAM and a single virtualised disk split into three volumes spanning a single partition. The volumes host the operating system, the FDFs and their respective data, and the IIS webroot, mounted on C:, D:, and E: 'drives', respectively.

The VMs are allowed to run for 30 minutes collecting information about the systems whilst under light load. Light load is defined as one web-service query per 30 seconds, on average, sent via cURL. This behaviour is adjusted slightly via the randomisation of the starting time using a 'scheduled task' on an external system outside of the virtual environment.

During this time, the performance tests are evaluated once every 60 seconds. If a system passes all the tests, each respective FDF saves both the configuration and metric data it gathers along with an XML schema file to a local data store. All of this information is stored in a sub-directory called 'data', which is located based on where the application currently resides. These files serve as a mechanism for loading saved configurations quickly and, as a consequence, allow for likeness testing and comparison using the same inferential data.

The experiments themselves operate by comparing the effectiveness of the FDFs through accuracy, precision, prediction time, fault position, the number of leads, and confidence values. To achieve realistic samples, the FDFs are injected with two types of faults.

The first type of fault is configuration based (ACCs), whilst the latter is instantiated by hard-crashing specific process threads (DFIs). The former consists of shutting off services or making changes to the system using normal administrative methods. These changes are made in such a way that they are expected to intentionally generate faults. This includes changing disk structures, service states, and other properties that administrators would normally have access to. The latter consists of copying incorrect instructions directly into the address space of another process, which in turn is expected to produce a controlled crash violating one or more performance tests (Table 4.2).

The ACCs include: Disabling the network card, disabling the W3SVC service, removing the volume upon which the IIS webroot is contained, removing all free space from any of the three volumes, and disabling network access from outside the virtual machine's local purview. The DFIs we instantiated included crashing various services such as the Windows IIS 7.5 W3SVC and Domain Name Service (DNS) service, and the IPv4 network stack.

Each ACC or DFI is run six times on the same FDF using 5, 10, 15, 20, 25, and 30 configuration samples. This allows for the realisation of trends within each approach, and to see differences in both output and FDF confidence during each specific test.

To keep the results viable each fault is induced using the same steps, and in rapid succession. Due to hardware limitations and problems estimating resource allocation in cloud environments, the results of these experiments are not run concurrently. Thus, although using the same data to populate the FDFs, there exists some amount of *drift* between tests.

It takes about two minutes to execute all six tests in either the DFI or ACC conditions using scripts and VM snapshots. This is assumed to be an acceptable amount of time, so long as the FDFs are retrained with fresh data after each series of fault injections.

These steps are particularly helpful when trying to understand performance and effectiveness

of the FDFs. It minimises external factors, whilst highlighting the properties inherent to the stochastic primitives, such as learning algorithm, training time, and memory usage.

Critically, the systems train their stochastic primitives at different times. In the case of HMMs and ANNs, primitives are discarded and retrained on every successful Performance test evaluation. This requires an active service on the VMs to parse the data in rapid succession – it also presents an upper limit to the amount of data that can be gathered. This limit is mitigated in the RBM implementation of the FDF by using lazy parsing. The primitives in the latter are trained only when a fault is detected via a performance test failure.

Once the primitives are trained they are used to generate potential root causes (*i.e.* leads). Leads are generated by examining the behaviour of the primitives sampled by the FDFs. As each primitive is sampled, a vector is created that indicates whether or not the previously observed value is identical to the value sampled at that specific time interval. The result is then used to train a stochastic primitive that is assigned to a specific feature – either upon fault detection or as soon as the data is ingested. This means there is one primitive per feature provided by the WMI framework.

Measuring the results of the FDFs is done using the traditional metrics of *true positives*, *false positives*, *true negatives*, and *false negatives*. True positives are determines when the correct fault has been identified, whilst false positives are any faults above the ‘correct’ fault in the ordered list of fault hypotheses – should it be present. Conversely, true negatives are determined when a fault is not detected and is not expected to be present, whilst a false negative is when a fault is expected but is not detected or identified by the system.

Due to the nature of false positives and false negatives, these metrics are evaluated by hand. This is expected as there is no way, by definition, for the application to detect such a state without external validation. It is also the reason why faults are injected in this experiment with the source already being known.

The confidence values for each approach are generated using different methodologies based on their respective learning algorithms. In the case of HMMs, the confidence value is provided natively using the Baum–Welch algorithm based on the strict probability of the likelihood of the suspected faulty feature’s last N behaviours, where N is between two and the specified windows size (*i.e.* 30). ANNs from these experiments use the simplest of prediction metrics: Naïve Bayes. The last observed state is assumed to be heavily weighted towards the expected behaviour of the feature’s next state. RBMs use the same approach as the HMM approach, with the exception of the accompanying learning algorithm being CDL.

Use of vector analysis to forecast the likelihood of a feature's behaviour in Naïve Bayes requires setting a minimum confidence value – in this case, 80%. This is done as there is no reinforcement learning from which the approach could dynamically weight its expectations. As such, the last observed behaviour in a known good state less the probability of change for the last number of up to 30 samples is utilised. Any feature behaviours that are not predicted to or do not meet this 80% threshold are ignored. In all other cases, the learning algorithms are not provided a minimum threshold.

4.7 Comparison & Inference

The FDFs and UBL operate in similar fashions. Their behaviours regarding how information is collected, parsed, and analysed is comparable, as are their intended purposes of reporting specific features suspected to be in error. However, their implementations and design assumptions differ significantly. To understand the relative impact and successes of these approaches they need to be compared.

The technical differences in how the FDFs and UBL collect, classify, and analyse information are described in this section. Specifically, the number and types of features being monitored, and how the collected data is used, stored, and evaluated are examined. These properties are discussed before a traditional analysis of the self-healing frameworks' prediction (*i.e.* planning) and recovery strategies – including what learning algorithms have been implemented, and under what assumptions.

4.7.1 Baseline Establishment

The following section describes the details for establishing a baseline and comparing results between approaches, and discusses the application of the hypothesis described in 1.2.1. It consists of three experiments, each using different implementations but adhering to the same overall logic: Training stochastic primitives with observations of feature behaviours to use forecasting as a mechanism for identifying the root cause of a fault.

The first two experiments are original works that compare simple stochastic primitives (ANNs, HMMs, RBMs) using Naïve Bayes, Baum-Welch, and CDL learning algorithms, respectively. The third experiment focuses on testing the stochastic primitives using a SOM that is trained via Euclidean Distance comparisons. The first two experiments are original works that have

been written, designed, and implemented by the author of this book. The third experiment is of external origin and used as a basis for comparison for the state of the art [28].

Each subsection provides implementation details for the aforementioned approaches. It discusses what tools are used, how the primitives are implemented, and provides a brief theoretical background. The final subsections describe the physical implementation of the first two experiments, and compare the external approach.

Results from these approaches are discussed in the following chapter.

4.7.2 UBL - An External Basis for Comparison

The ability to observe and learn from historical behaviours is not unique to the FDFs or, for that matter, to self-healing systems. The same approaches can be said to be at least tried in security research [103], amongst other areas. However, the ability to predict and forecast features to find faults is somewhat more limited when considering the use of unsupervised learning.

Only one additional known external example of this approach exists in self-healing systems research – UBL [28]. UBL is a cloud-based approach for identifying faults by looking for unexpected data in feature behaviours. It leverages a special type of ANN called a SOM. This primitive reduces relationships between feature sets into a two-dimensional lattice. Information contained in the lattice is traversed using neural weights which then model a system's behaviours. It is this modelling that allows for the forecasting of a system's feature data.

At a high-level, UBL operates similarly to the FDFs, however it has several fundamental differences in how information is gathered, classified (*i.e.* labelled), and forecasted. Like the FDFs, UBL periodically observes feature data which it then converts into vectors to train a stochastic primitive. Once the vectors are created differences begin to emerge in how that information is used. Instead of forecasting behaviours retroactively, UBL actively predicts feature behaviours so that it can preempt faulty states.

UBL uses a small, custom written application (*i.e.* daemon) to monitor feature changes in a fashion similar to the FDFs. Once instantiated it interfaces with and reads data from two sources: The Xen 3.0.3's *Domain 0* and the */proc* file system. Each gives access to behavioural data associated with features – such as changes in free memory, disk I/O, and other, similar properties.

Data is polled once a second before being used to either update the SOM's learned behaviours (*i.e.* neural weights) or to determine if a fault is present. Learning in UBL occurs under two

conditions: A ‘bootstrap’ phase is used to instantiate the primitive, after which a continuous update mechanism is present to adjust neural weights with current operational values. In either learning case, the neurons’ adjustments are made through a learning coefficient that manages how fast the primitive updates its neural weights.

The labelling of data is done through this initial phase. The initial training data is assumed to be valid, and all other data is evaluated based on the results of the bootstrap phase.

The bootstrap phase occurs until each neuron within the SOM has updated their respective neural weights a total of 10 times. In this case, the SOM’s dimensions are 32 x 32 indicating a total of 1024 neurons. Each neuron maps to an individual observed feature – therefore, all features must have active data or the SOM will never complete training. This is an important distinction as features with low behavioural frequency cannot be monitored using UBL.

Which neuron is selected for learning occurs by comparing the Euclidean distance of a sampled input vector with each neuron’s weight vector within the SOM. The neuron with the smallest distance value is then used. Estimated times from the original work for initial training are between 42 seconds and 7 minutes, depending on what percentage of the CPU is used for this task.

The second type of learning is done incrementally whilst all neurons are in a ‘normal’ state. Neural updates in this manner, however, have caveats and a deterministic operating period exists for each neural weight. Dean, et al, describe this as a problem of convergence:

> *... too many incremental updates may degrade the quality of the SOM as all weight vectors may converge to a small number of vector values. This can happen when the system starts to process a completely different new workload.* [28] [p. 3]

UBL addresses this problem by reinitialising the SOM and then using another subsequent *bootstrap* phase. There are potential implications to this methodology, but they are discussed further in Section 4.7. Once the SOM is trained it can use the Manhattan distance of neighbourhood size between neurons to forecast extreme shifts in feature behaviours.

Manhattan distance is calculated based on the distance between a neuron and its immediate orthogonally adjacent neighbours within the SOM. This metric is referred to as the *neighbourhood area size* in the original work. Anomalies are determined via this metric’s size:

> *If the neighborhood area size is small, we know that the neuron we have mapped to is in a tight cluster of neurons, meaning the neuron is normal. On the other hand,*

> *if a neuron maps to a neuron with a large neighborhood area value, we know that the neuron is not close to other neurons, and thus, probably anomalous.* [28] [p. 4]

Unfortunately, specifics as to what constitutes the range of these values is not explicitly given. It is inferred, however, that the values described herein are relative. This is due to the fact that all collected data is normalised within the SOM.

The Manhattan distance is used to predict the performance of various features within the SOM. If the neighbourhood area size is large, then the UBL framework attempts to make a multi-step ahead prediction as to the trajectory of the neuron being observed. This is referred to as the *Mapping Phase*.

Prediction capabilities during the Mapping Phase are divided into three primary categories: *Normal*, *Pre-Failure*, and *Failure*. These are notably different groupings than those used in the FDF approaches which operate in a reactive fashion. The first is an indicator of expected values given a threshold for the area size. The second indicates a larger than expected set of values, and a trajectory exceeding the maximum threshold limit; the latter indicating a failure state.

Using the aforementioned logic, UBL continues to operate until it either encounters a pre-failure or failure assessment for one or more neurons in the SOM, or until it reaches a convergence due to overtraining.

UBL is implemented using Virtual Computing Lab to emulate Amazon's EC2 infrastructure and operational properties. Each VM runs on top of one of five machines running Xen 3.0.3 on a 64-bit version of CentOS 5.2. Each physical server, in turn, manages five virtual instances. The polling information via Xen's Domain 0 interface is done through two libraries: libxenstat and libvirt.

Although training is done locally on some of these instances, UBL has the ability to train SOMs externally of the machine upon which they are running by transferring the vector state information to dedicated learning systems – machines that are dedicated to building UBL specific SOMs. Since these machines do not have external responsibilities, they can fully dedicate available CPUs to training of these primitives without performance impacts.

SOMs need to be instantiated using not only the bootstrap phase, but also using cross-validation from existing feature data for training. This is to avoid problems where some SOMs "*would only represent a subset of training data values*" [28] [p. 3]. Otherwise, the result is partial training of a SOM where some features are overlooked and their corresponding neurons remain untrained.

Faults are instantiated through a variety of stress and performance testing suites under several

application platforms including RUBiS, IBM System S, and Hadoop. Each application platform leverages 2 or more testing suites to induce faults via high-frequency resource utilisation – a pattern that is in keeping with UBL's expected inputs. The testing suites are *Bottleneck*, *CPUHog*, *CPULeak*, *Memleak*, and *NetHog*. Not all application platforms are subject to the same tests, however – with the exception of *Memleak* (Table 4.3).

Testing Suite		RUBiS	Hadoop	System S
Bottleneck	►			•
CPUHog	►		•	•
CPULeak	►	•		
MemLeak	►	•	•	•
NetHog	►	•		

Table 4.3: UBL Testing Suites. UBL uses various testing suites on a number of application platforms. Although some of these tests are used across all scenarios, the majority are not. For simplification, a testing matrix is included here.

Tests are administered "*between 30 to 40 times*" [28] [p. 5] before their comparisons against Principle Component Analysis (PCA) and κ-Nearest Neighbour (κ-NN) are displayed using receiver operating characteristic (ROC) curves. Some results are smoothed using between 5 and 50 points in addition to their existing normalisation.

4.7.3 Collection

Information collection for both approaches happens either through the use of local daemons or via API interfaces. These interfaces act as controlled, authenticated gateways into a host, and as a mechanism for formatting the returned data. Their implementations differ, however, in both their fidelity and their use. Still, each framework operates by interfacing with a common point for sampling feature data.

In both FDF instances, the features and attributes within the aforementioned classes are catalogued at a rate of once per minute then stored locally both in volatile and non-volatile memory. Each collection consists of datasets, tables, and tuples parsed into binary vectors or raw configuration data stored as XML, respectively. Storing information in XML files is used for resuming the service when running identical fault tests under variable conditions, such as using fewer configuration samples.

Once data is collected it is then stored in an intermediate state – how and when depends on the implementation. For UBL, once the sampled information is collected it is used greedily to

train a SOM either locally or via a separate VM. This behaviour occurs continuously and is not limited to a specific time-span or window. Eventually the SOM either predicts a fault or a training convergence occurs. The FDFs are trained either greedily (HMM, ANN) or lazily (RBM), using windowed datasets of the user's specified size in minutes. All approaches build vectors out of change data between samples to form the basis for later analysis.

4.7.4 Classification

The information gathered by these approaches consists of unlabelled performance metrics and configuration data. However, using this information to decide on the source of a fault first requires that this information be accurately classified – a non-trivial problem.

The state of the art for autonomously and accurately classifying unlabelled data in general is outside of the scope of this chapter. However, on occasion, problems in classifying unlabelled data are discussed in brief and as needed. This includes the strategies implemented by each of the aforementioned approaches, and their respective limitations.

There are a number of differences in how the UBL and FDF approaches classify data – from how much data is utilised, at what point the information is classified, and both how and when data is processed. These differences are associated with the relative uses of each framework although some properties are based on assumptions.

For example, the training phase for UBL is tested *in situ* before being applied. Using a training phase provides an advantage in that it does not require a specific set of performance tests or roles to be provided before classifying data. However, using performance tests follows a common tenet in self-managing systems research – the ability to provide high-level policies to systems as a primary form of administration. It also allows for the specification of specific areas of interest – an approach that can reduce false positives.

The UBL and FDF implementations both use unlabelled data to forecast anomalies by predicting unexpected changes in feature attributes. Predictions are made by observing a period of known or assumed *good* states to train primitives in order to recognise an expected set of behaviours. Once this training is complete, observed data is then classified heuristically into one of several states.

UBL's three classifications for state are determined by calculating the Manhattan distance of a neighbourhood area size using individual neurons. By analysing the differences in neighbourhood area size UBL is able to classify the behaviours of individual features as being

in one of three aforementioned categories.

The FDFs classify information into only two states: *good* or *faulty*. However, rather than labelling the behaviours of individual features, the entirety of a system's configuration is given a classification before looking for feature changes using performance tests. Performance tests consist of operational validation criteria – properties that indicate a system is operating within its intended role. If any performance test fails, the entire set of features sampled is classified as *faulty*. Once a classification is made, the data is then parsed using either the previously mentioned greedy [18] or lazy fashions [19], respectively.

Using three states offers the benefit of proactive analysis over reactive analysis. Although the intention is to establish a solid understanding of accuracy in the detection of faults, doing so proactively has several potential benefits over the reactive solutions presented here. However, it may also be moot to need to address problems proactively under any of the following conditions: the fault detected cannot be prevented, the fault manifests too quickly for a solution to be implemented, or an evolutionary approach is used to prevent further instances of said fault(s). In several instances, the second condition impacted UBL's results and performance.

Instantiation of classification properties are different between UBL and the FDFs. Both experiments expect the systems to start off in a healthy state for the purposes of initial training. The training then forms the basis of analysis for the FDFs, but for UBL it also is the primary component for classification of sampled data. This is because the initial neighbourhood area size is calculated within this training period and thus where all other data must be inferred. Additionally, as the data is not windowed, this is a static property outside of small, incremental updates that effectively amount to an average of values.

How long the data is stored and how it is ingested plays a critical role in classification. Using a windowed approach for information parsing allows for the avoidance of convergence in training data, and greater adaptivity to changing environmental variables. Both of these properties represent advantages in implementation but they come with a cost. Windowing necessitates more memory and post-processing requirements as purely additive measures are no longer sufficient. As such, the expectation is for windowed information to take longer to classify and process. This is seen readily in the results of both approaches, independently.

The way data is ingested impacts when and how classification occurs. WMI provides attributes associated with features in a semi-structured, non-uniquely identified tuple. In order to address removals of devices and multiple features that share a similar namespace this exigency must first be addressed. The FDFs use of a dictionary as a WMI class unique identifier serves this purpose, but also necessitates much slower parsing then direct observation to vector conversion

– as occurs in UBL.

Xen samples metric data from a number of different features. As the values within these samples have multiple ranges, their relative performance and consequent classifications can become difficult. UBL's solution is to normalise this information to unilaterally use the same evaluation techniques across all features. This reduces the fidelity of the content, but lowers the programmatic overhead needed to classify the sampled data. The FDFs comparatively have greater fidelity as they do not trim or normalise their respective results.

The classification of data happens at the feature and system levels for UBL and the FDFs, respectively. This distinction impacts other aspects such as frequency of data collection and the number of features and attributes sampled. It also affects how the data is analysed: There is an implied relationship between the number of observations and what predictions, if any, can be made. However, a larger number of observations does not always provide for more accurate results.

4.7.5 Learning & Analysis

The categorisation of feature data allows for the training of primitives and, if necessary, subsequent fault analysis. However, in order for a self-healing system to correctly identify the potential cause of an fault it must first determine if a fault is present.

How each framework trains and detects potential abnormalities in feature behaviours varies, both in assessment and implementation. The primitives used, what learning algorithms are associated with those primitives, how much and what type of training data they receive, and for what period of time are all variable with different values in the aforementioned experiments. To understand the complex relationships between these variables some basic tenets about how these frameworks detect and isolate anomalies is given.

Fault detection in these self-healing systems occurs after data has been collected, but during either categorisation [18] or when collected data is being parsed [19, 28]. In either case the categorisation behaviours occur before an analysis is made. These stages consist of multiple milestones including initial data collection, conversion into vectors, and their subsequent assimilation into one or more stochastic primitives.

Each stage of this process is important as failures at any point will lower the capabilities of the frameworks. If a failure occurs in detecting fault conditions for categorisation, the learning capabilities of the primitives will be lowered – or potentially even rendered moot.

There are two primary stages associated with the use of stochastic primitives in self-healing systems. The first stage is provisioning and instantiation. This includes the initial build of the object, and its subsequent training. The second stage is the use of that object – typically to either feed in a vector and understand if it is associated with patterns that have already been observed, or to provide it with an output and synthesise a vector as in the case of CDL.

Fault injection evaluates the effectiveness of the latter stage. How much data has been observed, the learning rate, and the resource utilisation are just some of the key properties. As the variables within each of these areas are large in number, keeping the tests consistent is important. There is limited use in providing data without a control in this respect.

UBL implements its stochastic primitives differently than the FDFs. The authors describe it as starting out by instantiating a single SOM that consists of 1,024 neurons that are arranged in a 32x32 lattice. It then populates each neuron with a randomised weight. This weight is what determines the paths vectors take as they traverse through the SOM. Any path that meets an edge in the lattice can run into problems as the evaluation may be incomplete as distance is a factor in accuracy in terms of this traversal.

Afterwards, a *bootstrap phase* is used to train the SOM using historical data from the IRCache project (*circa* 1995) – this continues until each neuron has their respective weights adjusted 10 times. The reasons for using these values or this dataset is not explicitly given in the original work.

Once the SOM is trained it begins periodically sampling the system for further data via Xen's 'Domain 0' interface. The data is then either updated into the SOM or a differential analysis is performed using the Euclidean distance of an input measurement vector against each neuron's weight vectors, respectively. The weights in the SOM are then updated incrementally every time a sample is provided.

In this case, a sample is provided every 60 seconds. When the update occurs, each neuron in the SOM has its weights validated via a neighbourhood area size calculation using the summed Manhattan distance of each of its neighbours.

The FDFs take a completely different approach to instantiation: For each feature being observed, a separate primitive is built and trained. Like UBL, these primitives are built with default weights. In this case all objects consist of three layers, and randomised weight values – there is no cross validation. Once the primitive is ready one of several training use case scenarios exist. Broadly summarised, they can be viewed as being under the greedy or lazy data consumption models.

Any system using a greedy model will train the primitive immediately. If the primitive currently has learned less than the total number of maximum samples, that information is simply added to that primitive's existing knowledge. If the maximum number of samples has been reached, the primitive is destroyed, and a new windowed subset of information is built. Using this new information, a new primitive is instantiated and trained to take the old primitive's place.

Systems using the lazy model simply aggregate training data until a fault is suspected to have occurred. Once detected, a primitive is built for each feature, trained using existing data up to the maximum sample size, and then an analysis is run. This saves total computational resource usage, but slows down result generation due to the lack of pre-computation.

In either case training is dependent on several factors. In addition to primitive type, how long it takes to train the FDF is related to how many cycles (*i.e.* epochs) are specified, and the total number of features in question. In this case, 5,000 epochs are specified for RBMs, whereas the other two types of primitives are left to train only once per provided sample.

Once the primitives are trained they can forecast either single (ANN, HMM) or multiple points of behaviour (RBM). This ability is gained from the predictive reasoning capabilities of the algorithm along with the physical structure of the primitive(s). In the case of RBMs, for example, an undirected graphical model uses an approximated gradient for the log-likelihood of a specific behaviour. This is sampled using a Markov chain which is weighted towards the last observed state.

For reasons of scope and complexity, the functional and operational aspects of the respective learning algorithms are not addressed here in detail. The details of these algorithms are generally agreed to be well documented and readily available. However, CDL remains an evolving and not entirely understood methodology [13, 99, 95].

Fault detection is the primary catalyst between learning and analytical behaviour for all of the aforementioned approaches. The FDFs handle fault identification through performance tests; a simplification of expected system operating behaviours. UBL's approach however is much more involved.

The total Manhattan distance metric is the primary indicator of both fault presence and source. Neurons that have a small area mapping are assumed to be operating normally. Those neurons with larger distance spreads are indicators of either precursors to potential or existing anomalies depending on severity. The threshold for making these determinations is not explicitly given in the original work, but a 50% increase over the example value is given as an indicator of a pre-failure neuron.

The source of a fault is determined using the preserved geometric positions of each of the neurons within the SOM. Mapping neurons' expected behaviours with those near to the neurons suspected to be in an anomalous state provide an updated distance measurement from which to evaluate the cause. The inference between the feature and its associated neurons provides an avenue for identification. UBL's low level analysis has advantages in that it does not require roles or additional pre-requisites to be supplied outside of the bootstrap phase before becoming operational.

The FDFs fault analysis approach is similar to UBL's. Both approaches look for differences in observed and current behaviours with the context of expected performance benchmarks. The difference between UBL and the FDFs is that analysis is done in the latter based on forecasting specific feature behaviours.

Once a system's configuration is labelled as having not passed its performance tests, a set of stochastic primitives is made ready via the known *good* configuration samples. This represents a control from which to test the current *faulty* configuration. A differential analysis is done to see what features changed between the last known good sample, and the current fault sample data. Any features that show a change are short listed, and their behaviours are forecasted using only the known *good* data.

This information is then compared to the actual data present in the faulty configuration set. Once a forecast has been made a confidence value is provided via the learning algorithm that indicates the likelihood of that forecast being correct. Forecasts that are sorted in descending order based on likelihood (confidence) as an indicator of the potential root cause of a fault. Data produced by the application is then either passed on to an individual for evaluation, or can be used to hand off data to a third party application.

Perhaps the most critical point in evaluating these frameworks is how faults are injected into them. In both sets of experiments, a range of software suites are used to induce anomalies. These suites focus on performance criteria and volumetric approaches, as is the case with UBL, or controlled thread crashes and adverse configuration changes.

Each suite of approaches offers a mechanism for testing against conditions that are considered undesirable specifically in professional computing environments. However, these tests are performed equally in the UBL experiment . The technical aspects of these testing suites are covered in Tables 4.3, and 4.2, respectively, however, comparisons of their respective approaches are not.

UBL emphasises a performance based model to understanding feature behaviours. The testing

suites it uses emulate this perspective by forcibly injecting data into a system's services. The results manifest at different observable levels within the machine itself, including raw resource utilisation – such as memory, or CPU – and via service controls. An example of this is running Reduce functions through Hadoop until an abnormal signal is returned. However, not all tests are run across all instances (Table 4.4).

	Hadoop		**System S**		**RUBiS**	
Testing Suite	TP	FP	TP	FP	TP	FP
Bottleneck			*G*	*G*		
CpuHog	*G*	*G*	93%	0.5%		
CpuLeak					*G*	*G*
MemLeak	*G*	*G*	98%	1.7%	97%	2.0%
NetHog					87%	4.7%

Table 4.4: Summary of Testing: UBL. In the original work, results for UBL are presented textually, graphically, or sometimes not at all. *G* = Data via Graph Only, Blank = No Data, TP = True Positives, FP = False Positives.

As mentioned previously, there are two types of fault injection mechanisms within the FDFs: ACCs, and DFIs. To understand how each of these approaches operate and their criteria during testing they are expanded upon here.

ACCs operate by inducing valid inputs into the system in such a way as the result is expected to cause a fault. This is to understand how user error might be caught by the application in comparison to when a system's internal logic or other factor fails outside of human control. Faults in the ACC category are triggered by running scripted commands against the system using included service controls only.

DFIs handle this second category by inducing controlled thread crashes. This second type of error is the more traditional focus in prior art, but it is not the only issue worth exploring. Anecdotal evidence suggests that change controls made by individuals in professional environments are difficult to perform consistently – particularly when done by hand. Understanding the differences in a self-healing system's ability to differentiate between the two is therefore valuable, *a priori*. DFIs are instantiated through the use of process termination signals, random pointer walks within specific threads, or the use of a hex-editor to write (*w*) faulty, *no-op* / *eax*, (0x90) instructions.

Testing suites within the FDFs operate under a similar philosophy as those in UBL in that their results are specifically tested against operational performance metrics. In this case, the change of a specific feature's behaviour by the test administrator is expected to induce a state that will

cause a fault. This can occur under rather obvious assumptions – such as directly shutting off a service – but this is not always the case. For example: An external router is disabled under one of the FDF tests – a property not monitored by the frameworks – and a root cause of loss of network throughput is indicated. This is further discussed in the following chapter on results.

4.7.6 Comparison Constraints

There are some constraints and implementation details that differ between the FDF instances and UBL that are worth mentioning explicitly. Understanding and detailing these properties before discussing the results of each approach offers greater context to the information being provided.

The FDFs operate under similar conditions and principles to UBL. However, instead of focusing exclusively on feature behaviours, a combination of systems validation (*i.e.* performance) tests are used to provide context to changes in attribute data. Contextual information offers clues as to what shifts in attribute behaviours are expected or unexpected by attributing their values with SLOs. If all SLOs are successfully being met, then the configuration is given a context of being in the aforementioned *good* state. In this case SLOs are incorporated into and represented by performance tests.

The SOM within UBL can update its weight incrementally but not indefinitely. In the current implementation there is no way to expire old data as it is immediately incorporated within the SOM and then the sample is expired. Since the information cannot be expired the SOM's weights will eventually converge. This eventually creates an inability to perform the differential methods needed for classification and analysis.

The feature data examined by UBL is normalised into a range from 0 to 100. By examining both the minimum and maximum possible values of a feature before executing the number of neurons required drops substantially as their use is tied to the maximum and minimum values of observed features. However, as mentioned, normalisation of values may produce a drop in the fidelity of information. For example, if the change in the neuron's actual value represents less than 1% of its total possible value, then the change may not occur within the SOM. In some instances the original work states that normalisation values have also been reported to be over 100. These incidents are claimed to be non-impacting, but are not completely understood in terms of implementation or how they might influence the analysis of anomalies.

One of the primary differences between this work [18, 19, 20] and that of Dean, et al [28]. is that entropy in feature behaviour is not ignored in the FDFs, nor is any data normalised. These

are important distinctions as more data is handled in the FDF approaches in terms of the total number of features than in UBL, and that the potential for identifying the root cause of a fault can be truncated under normalisation conditions.

Forecasting behaviours is not an exact process. In each instance the abilities of the primitives are constrained by mathematical problems – such as the accumulation of error – but also the additional restrictions of their learning algorithms. This includes accepted and expected error in probabilistic learning, and numerous other factors.

Before being able to forecast behaviours a lead time is required in all of the aforementioned experiments. The amount of lead time and the level of accuracy are topics discussed in the Results section. However, there are some minimums that are present in each approach. A minimum of two *good* samples must be provided before a failure can be analysed by the FDFs. This is similar to UBL's 10-update per neuron pre-requisite, but with the added difference that FDFs use polling intervals at fixed time differences. This allows for some measure of prediction as to when training will be complete for the FDFs versus UBL.

Using performance tests may have some advantages in specificity, but it requires greater initial human oversight. The FDFs emphasise a high-level approach to determining both the presence and source of a fault. They are designed to automate alerting procedures and operate as independent services running in large-scale, centrally managed computing environments. However, requiring a set of role-based performance tests means that they operate less agnostically than UBL.

By looking for general areas where problems may exist it stands to reason that correctly detecting a fault that is associated within such an area is more likely – so long as the tests and analysis logic use the same points of reference. For example, a network connectivity performance test may help indicate which features are more likely to be the source of the fault assuming the context of the test is incorporated into the analysis logic.

Training RBMs is relatively expensive computationally. Compared to the iterative updates of UBL and the first FDF framework, the second FDF approaches are particularly intensive. Each RBM requires thousands of training cycles before being utilised. If there are a large number of differences in attribute states between the last known good and fault configuration samples, it could take several minutes for a potential root cause to be proffered by the application.

There is a certain amount of elasticity in an FDF's ability to forecast feature behaviours. The size of the training sets – both with respect to the number of features being monitored and the total number of observed configurations – influences the processing time, adaptivity, and

accuracy metrics. By increasing the frequency of the polling interval, the FDF's maximum window is constrained to a shorter time period. Once the maximum number of samples is reached, old data is discarded. Consequently, the ability to forecast data becomes restricted to a small subset of information. Increasing the maximum number of observed samples used for parsing comes with higher resource constraints, but it also provides more stable predictions and less sensitivity to outliers. Expiring old information helps to retain the correct scope from which to draw conclusions and avoids problems in over-training and convergence.

Lastly, different learning algorithms provide different results – which is both why they are interesting and why they are difficult to compare. It is not always clear as to how factors impact each other, or if there are relationships between attributes either in the learning algorithm or the data when a fault is detected. However, some of the results suggest that it should be possible to help determine the existence of such relationships. ?

In summary, both the FDFs and UBL look for feature changes to determine the root cause of faults. However, in the case of the latter a different primitives and learning algorithms are used alongside different tests: FDFs use the same tests for each experiment, and UBL uses different tests per the authors' selected performance testing suites. Additionally, both of these experiments evaluate slightly different conditions – both are looking for faults, but UBL looks for changes in performance metrics outside of an expected range that is dynamically generated, whilst FDFs look for violations of SLOs via performance tests. Nominal other differences remain as summarised in Table 4.5.

Property	UBL	FDFs	
Primitives	ANN (SOM)	ANN, HMM	RBM
Learning Algorithms Tested	3	2	2
Training Cycles	Conditional*	5–30	5,000
Windowed	No	Yes	Yes
Polling Interval	1 second	60 seconds	60 seconds
Forecasting	Multi-point	Single-point	Both
Ingest Type	Greedy	Greedy	Lazy

* Training in UBL occurs randomly until each neuron has been updated ten times.

Table 4.5: FDF and UBL Property Comparison. This table presents a summary of the comparison of operating properties in unsupervised self-healing frameworks discussed in this chapter.

CHAPTER FIVE

RESULTS & DISCUSSION

The FDF experiments demonstrate the ability to accurately identify the root cause of faults by using a combination of unsupervised learning, performance tests, and stochastic primitives. Most results are show support for the hypothesis and overall the work in this thesis helps to enable the autonomous identification of the root cause of errors.

This chapter provides the results from each FDF experiment whilst contrasting them with an external approach called UBL before providing a discussion and leading into conclusions.

5.1 Introduction

This chapter presents the findings from the two separate but related FDF experiments discussed in the previous chapter. The first experiment focuses explicitly on ANNs and HMMs whilst the latter explores a nearly identical implementation using RBMs. The difference between these experiments is that the primitives are not trained until a fault is detected when using RBMs. Switching from a greedy to a lazy ingest mechanism provides notable changes in some of the timing and list size results, but otherwise all aspects are identical including what kinds of faults are injected, how they are induced, and the volume of data used to train the primitives.

The results of these experiments are described sequentially as they are contrasted with an external experiment (UBL). This guides the interpretation of both series of results from the FDF experiments along with an external measure for comparison. A discussion section then provides a degree of synthesis in these findings before outlining future areas of exploration and conclusions in the following chapter.

5.2 Overview

This thesis partially answers the research questions as described in Section 3.1.2.

Chiefly, the experiments explore the degree of human interaction required to accurately find the root cause of a fault by using unsupervised learning with stochastic primitives (Question 3), the impact of information fidelity on such an approach (Question 6.c), the impact of different types of faults on fault identification (Question 5), how quickly faults can be identified (Question 6.a), and the viability of examining only feature change data to accurately determine a root cause (Question 6.b).

More generally, the experiments emulate the IBM's proposed "policy-based" approach of managing systems [4] by implementing SLOs and general performance tests rather than a series of software unit tests (Question 4), whilst Chapter 3, separately, provides a comparison of self-healing systems frameworks' computing environments, management styles, and learning algorithms (Questions 1 and 2).

Finally, a comparison is provided directly between three distinct primitives and learning algorithms using the same experimental controls. The first two primitives form a baseline whilst the latter is intended to demonstrate capabilities of the approach. To verify these capabilities and understand relative performance, an evaluation against a similar, independent fourth experiment is presented using publicly available results [28].

Although other attempts at comparing the performance of self-healing systems frameworks have been made, such as with the aforementioned DTAC project [43], there does not appear to be much in the way of contributions in this capacity. This limited the scale of the comparison made, but a best effort was made and the results seem both usable and interesting (Question 7).

The evaluation of the experiments within this book focus on whether or not it is possible to correctly detect the presence of a fault and then identify its source using a combination of unsupervised learning algorithms and a comparison of actual and synthesised feature behaviour data. In each experiment, ANNs, HMMs, and RBMs are able to complete this task accurately and within not more than 15 seconds. In addition, the autonomous classification of feature behaviours by using performance tests is also shown to be an accurate approach for labelling data.

Although results are largely positive, there are some caveats that presently include a necessary training period between 5 and 30 minutes before the fault identification capabilities can be used, moderate to high computational costs, and a compulsory initial set-up by a human administrator

where a system's role is defined (*i.e.* the instantiation of said performance tests). These are hurdles that present future areas for improvement, but it is not believed that they deter from the validity of the results nor their contribution to the state of the art.

The approaches employed in the FDF experiments are also compared and validated against an external experiment – UBL. The findings from UBL offer support for the validity of the approach, and demonstrate some of the advantages the FDFs provide to the academic community over prior research. Examples include the importance of contextually relevant data, that proper labelling and training sets for primitives remain important factors to consider, and that synthesised, older or normalised information are all factors in guiding dynamic system behaviours.

A variety of potential improvements remain open for further study. Fidelity of information and entropy in data-sets represent two such areas – it is worth noting that both the FDF and UBL experiments run into areas of difficulty when high degrees of entropy in feature behaviour are encountered. However, difficulty establishing patterns of behaviour when such patterns are seemingly not present are to be expected. Alternative techniques such as cross-referencing the likelihood of multiple feature behaviours could help relieve some of these problems. This avenue of potential research and others are discussed in further detail in the following chapter.

5.3 Results

The results in this thesis came over a substantial period of time, through much process, and with concerted effort. Initially, the experiments described herein were intended to be iterated over a range of values concerning three primary variables: window size, sample frequency, and learning module. This was designed such that individual markers for the volume of data being used, the resource costs involved, and the accuracy of each of these properties could be evaluated holistically and explored through multiple iterations.

Although this type of experiment is still possible using the resources from this thesis, the initial creation and setup of the FDFs introduced some unintended resource constraints. This is a problem that is multifaceted in nature – from programming priorities moving from building a simulator to an actual, working application consistent with the properties of an independent framework, to ensuring accurate reproducibility in the results using academic and industry best practices and standards. Once these problems were solved, iterating the experiments became difficult to address with the remaining time constraints.

The creation of the FDFs occurred over two separate development cycles including a separate series of trials for gathering information on variance. In the latter case, an identical feature is selected that is related to the correct root cause of a fault. The first feature correctly identified with the root cause was selected. This feature may or may not be the same feature selected in subsequent trials as each fault may have multiple valid sources. This mirrors other approaches, such as UBL, allowing for easier external comparison. The sole purpose in the variance trial is to watch how one specific feature traverses the graphs.

The first FDF included building an ingest mechanism after deciding upon the correct tools for storing, parsing, and interfacing with feature data and took about a year of development work. This included writing up proper testing procedures so that results could be validated and building an ingest mechanism, global variables that could be adjusted trivially, a simplistic testing interface that addressed IBM's initial criteria, and the ability to replay and recall information from prior observations. This also includes designing tests for the FDFs. Although not explicitly noted in the literature review, it became clear that faults had different types of instigators – including human error. Testing the difference between detecting a legitimate fault and one intentionally manifested then became a priority.

Development of the second framework took an additional three months to validate, implement, and integrate new primitives and learning modules. The complexity of the RBM code necessitated additional development work for the Accord.NET and AForge.NET frameworks that was not anticipated. Specifically, test cases had to be implemented to ensure that the primitives were being populated correctly, and that the ingest of data was presented in the correct format using the same amount of data as in the previous experiment.

In the end, each experiment was run 30 times for each sample size subset. Subsets were divided into 5 minute intervals to keep the graphs readable and to highlight differences more clearly. The subsets tested were addressed as 5, 10, 15, 20, 25, and 30 minutes worth of data, respectively, to address the maximum window size. Further samples were not taken beyond the initial 30 minutes as they would have been expired and thus remained the same total size. The window size of 30 was decided upon based on anecdotal evidence based on normal operational testing windows – specifically my personal experience as an engineer, and loosely questioning peers in industry. It represents a normal time-span in which rapid traffic changes could occur in the monitored service, whilst still being short enough to be highly adaptive.

The number of runs was decided based on the central limit theorem. There seems to be some disagreement with using the value 30, however, and that the recommended number of samples can readily vary from 20 to 50. The value of 30 was decided upon based on a consensus and

discussion with peers within the Computer Science department of the University of St Andrews.

5.3.1 The FDFs

Is it possible to accurately identify the root cause of a fault using performance tests, stochastic primitives, and unsupervised learning?

These experiments demonstrate positive results and overall show support for the hypothesis under both conditions – where success is determined either by the fault being in the top 10 recommended features, or based on proximity to the 0^{th} index – for finding and identifying the root cause of faults based on feature change behaviour without human intervention.

However, although most of the results demonstrate success, some of the results in each FDF's performance did not meet these conditions and show potential for improvement. Overall, it takes about 20 minutes before getting consistent results that are useful for fault identification.

Concrete results were generated showing:

1. The number of computing samples within the FDFs directly impacted the amount of time it took to generate a descending ordered list of fault hypothesis. Using ~7,500 features, in implementations where a greedy algorithm was used each sample added ~150ms for HMMs using Baum-Welche, and ~90ms for ANNs using Naïve Bayes. Likewise, lazy algorithms using an RBM leveraging CDL being trained at 5,000 epochs added ~560ms per sample.

   This is important in understanding the computational cost of each approach, and how much time is needed per sample to generate leads if attempting a proactive solution.

2. Sampling the probabilistic likelihood of feature behaviours allowed for accurate determination of potential root causes as expressed via *confidence*. ANNs, HMMs, and RBMs, each provided accurate, descending ordered lists with the correct faults based on confidence. The proportional calculation used with the HMMs was an artefact that was not initially considered. Ultimately it did not allow for monitoring the progress of reinforcement in observations because all results were relative to the perceived most likely root cause (*i.e.* feature). Still, both FDFs produced usable, accurate results using this metric regardless of the primitive or learning algorithm used.

   Thus, when compared with the accuracy and precision results, concrete evidence is given that *confidence* is a good indicator of the learning algorithms ability to correctly detect abnormal behaviours in features, and that CDL in particular is an excellent learning

algorithm for this type of analysis.

3. The ability to indicate the correct potential root cause via *fault position* was clearly demonstrated, regardless the fault type, with RBMs being more successful over other approaches. This is indicated by the ability to generate descending ordered lists with the correct root cause more towards position zero than the other approaches in almost all cases.

   Specifically, at 5 samples, all approaches perform similarly – however, drops in fault position occur with the HMM and RBM approaches – notably without a decrease in accuracy. The rate of decrease in fault position is sharpest in RBMs, which showed an average decrease of about half a position per 5 samples. HMMs show a similar overall decrease, but not as consistently or sharply; at 30 samples this metric converged with RBMs. ANNs produced the least desirable results with the correct root cause oscillating between positions 4 and 5 regardless the number of samples provided. Highly chaotic feature data such as Free Disk Space provided limitations to the analysis abilities of the learning algorithms associated with the primitives, which was expected.

4. A smaller number of fault hypotheses (*i.e.* leads) occur when using greedy ingests, but using this approach costs more computationally, overall. The details of this conclusion are technical and discussed further in Section 5.4, but in summary the smaller subsets of calculations provide a mechanism for reducing the number of potential leads when compared to analysing feature behaviours at larger key intervals. For each good sample that was expired, the computations performed to understand feature behaviours were effectively wasted. A lazy algorithm avoids this pitfall in exchange for longer times at during fault identification (*i.e.* generating a list of fault hypotheses) at the time of fault detection.

5. A windowed data-set is a necessity to avoid problems with convergence. This is confirmed through the presence and suspected root cause of such a problem in the comparison study (UBL) (as confirmed by its authors [28]), and the notable absence of this problem in the FDFs. By intentionally running similar experiments designed to avoid this problem, a case has been developed that lends evidence to the mitigation of this problem.

The first experiment attempts to fulfil two primary purposes: to explore the validity of the approach, and to establish a baseline for understanding the effectiveness of this and future experiments. Before this experiment, no public studies were known to provide a direct comparison of stochastic primitives for finding and identifying faults within a self-healing system.

The measurements taken to compare approaches includes *confidence*, *total leads*, *time-taken*,

and *fault position*. Each of these measurements are defined in Section 1.4, but a brief overview is also provided near figures in this chapter, where appropriate. The values in the figures contained within this chapter are averages from each experiment.

Variance measurements are absent in the trials involving ANNs and HMMs. This is due to the fact that trials were executed using the same feature data via VM "snapshots" – literal byte-level copies of the systems being tested. Once an object was instantiated, given the same information under nearly identical conditions, it performed identically with the exception of minor (< 10ms) timing differences.

In summary, it is possible for both an HMM and an ANN to identify faults by generating an ordered list of potential root causes. However, using Baum-Welch and Naïve Bayes, respectively, did not produce particularly precise results, and their computational resource utilisation is higher than expected. Also, as expected, the HMM approach outperforms the simple ANN in a number of trials.

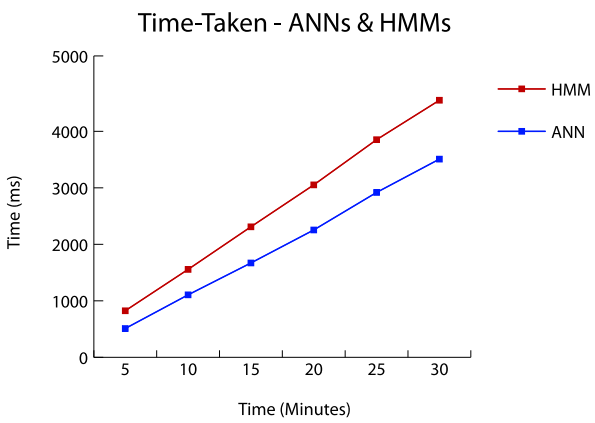

Figure 5.1: FDF v1.0 - Time Taken. Time-Taken represents the number of "*ElapsedTicks*" converted to milliseconds (ms) between when a fault is detected and the return of an ordered list of potential root causes. The ANN took less time than the HMM to produce an ordered list of fault hypotheses. Shortened times allow for a wider range of recovery solutions making them more desirable. Both values grow linearly per the amount of data being provided.

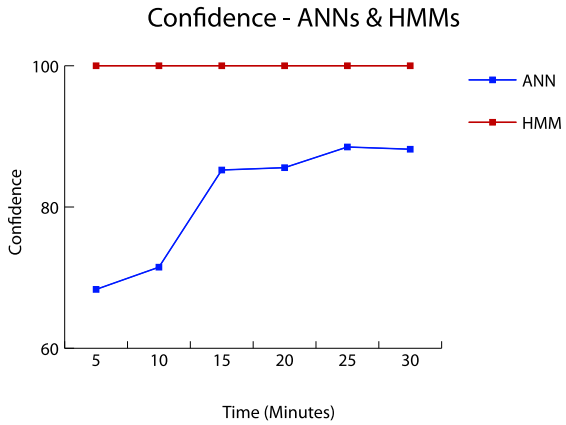

Figure 5.2: FDF v1.0 - Confidence. Confidence conveys how likely the FDF's suspect a given lead is associated with the correct root cause of the detected fault. Converse to the amount of time taken, the HMM produced much higher confidence values than the ANN. This is a result that was unexpected because of the way that the Baum-Welch algorithm calculates probabilities.

The time needed to train each FDF and how likely each a feature is believed to be the correct root cause can be seen in the time-taken and confidence graphs, respectively (Figures 5.1, 5.2).

Time-taken is a metric designed to provide a high-level understanding of the computational

resources required to obtain a result. It also directly relates to what constraints may exist relevant for forecasting feature behaviours. Longer time-taken implies a greater required lead-in time between when the fault is suspected to be manifesting and when a solution is implemented. Being able to detect manifestations of faults through forecasting allows for a shift from a reactive to proactive set of solutions, a related topic for future study.

In every case, the HMM required more time to provide a list of potential root causes than the ANN. These results are as expected as the ANN implementation is intentionally simplistic by comparison to the HMM. It also sets a baseline with averages between less than 1 second and 4.5 seconds for the diagnosis of faults.

Confidence values are a probabilistic measure of a feature's behaviour given some number of previously observed states. How this occurs is covered earlier in the thesis (see Section 1.4), but ultimately the value represents the inverse likelihood of an observed feature behaviour. This is evaluated by comparing the state of a feature in a faulty configuration when compared to a series of configurations that passed their respective SLOs. The less likely the change to occur, the higher the confidence value.

The difference between the ANN and HMM approaches is immediately obvious in terms of confidence. HMMs always return the most unlikely feature change in the sampled data as a value near 100%. This is an unexpected result, and as described in Section 4.5.1, make comparisons between Baum-Welch and other learning algorithms less conclusive than is desired.

Ideally, the results should show a gradual increase such that reinforcement gains can be tracked and then compared. This is observed in both ANNs and later RBMs producing expected results. As expected, ANNs in this experiment do not show a particularly fast or high valued reinforcement.

Interestingly, the type of fault impacted the degree of confidence. When the fault is introduced into the system via DFI, the ANN predicted the problem with greater confidence. Upon further investigation an interesting property is made evident when using ACCs: Controlled stops of systems touch multiple dependencies typically producing a larger number of correctly identifiable features.

In contrast, direct observation reveals a smaller number of noticeable feature changes under DFI conditions. Eventually, dependent features change under these circumstances but they seem to take longer to occur. The net result is that ACCs provide a greater number of leads for each FDF to investigate. This has some impact on precision as the evaluation criteria for these experiments

are focused on identifying a single feature (Figure 5.5).

Since all changes associated with the fault are equally weighted, diagnosing ACC generated faults is more challenging. Often several leads have the same confidence values and thus same fault position. Regardless, when attempting to ascertain the root cause the HMM is still able to identify the correct feature more often than the ANN. This is seen readily in the fault position results, and attributed to the more sophisticated Baum-Welch learning algorithm.

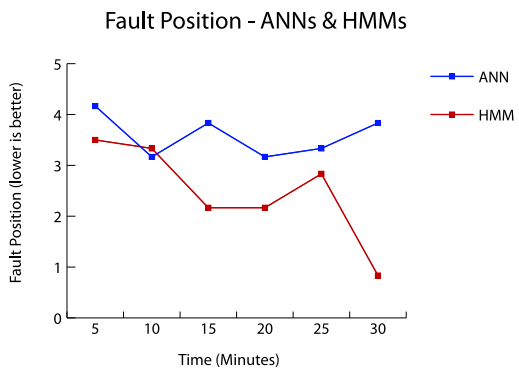

Figure 5.3: FDF v1.0 - Fault Position. The average position of a correct root cause as returned by the FDF is represented in this graph. As the lists are ordered by descending probability, lower values are better. The averages from the experiments show that the HMM outperforms the ANN in nearly every test. Additionally, fault position improvement is much slower with the ANN.

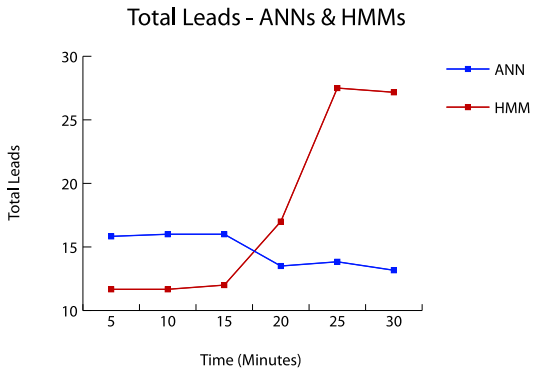

Figure 5.4: FDF v1.0 - Total Leads. FDFs generate leads when a fault is detected. This graph represents the average total number of suspect features (*i.e.* 'leads') at 5-point sample intervals. The FDF using HMMs is able to generate more leads than the one using ANNs, however more leads is not always better. The ideal result is a list containing only the features that are associated with the cause of the fault.

Fault position is a human validated metric based on knowing the cause of the fault, *a priori*, and monitoring the effects of either a DFI or ACC (Figure 5.3). It illustrates the FDFs' prediction of the correct root cause and is the primary metric for evaluating the precision and accuracy of the FDFs. Accuracy and precision metrics are directly related to how close the selected root cause is to the first index of the list . The lower the fault position value (*i.e.* index), the more correct the diagnosis made by the FDF. Again, the HMM regularly outperforms its ANN counterpart by frequently placing the correct root cause nearer to its 0^{th} index. This is the expected result based on the sophistication of the learning algorithms between these two primitives.

Converse to fault position, total leads is an autonomously generated metric that represents the total number of fault hypothesises generated by the FDF . Total leads is used to understand how many potential avenues for investigation are generated at the time of fault detection (Figure 5.4). This allows for an understanding of the performance of the FDF, and what correlated factors

may exist with other metrics, such as time-taken or precision. By understanding the reduction from the total number of features we start to understand the gains created by using the FDFs during fault identification.

Anecdotal evidence suggests that human administrators would find a singular or even three to five potential avenues feasible for use. However, list sizes with the correct fault ranging between 12 to 30 in the tests were not uncommon for both the ANN and HMM trials. Although this is an improvement over the 7,500 initial features, there is clearly room for improvement.

Additionally, the correct fault must be present otherwise the list size is not a useful measurement. In this experiment the correct feature is identified in most trials, however that may be more an artefact of the investigatory logic than the behaviour of the stochastic primitives.

In both the HMM and ANN based FDFs, changes in a feature's behaviour alone are not enough to trigger an addition to the list of leads. The changes must cross a certain threshold before they can be forecasted accurately and then evaluated as either expected or unexpected behaviours. This is where differences in the learning algorithms are most obvious.

The HMM is capable of setting a dynamic threshold for feature behaviours, whereas the ANN uses a statically assigned 80% value. The net result is that the HMM generates more leads when compared to the ANN. Although more leads can indicate greater sensitivity in detecting faults, higher values in this instance are not always better. In a perfect scenario only the exact root cause(s) are provided.

As mentioned, FDFs using HMMs are more likely to select the correct root cause than ANNs. Although selecting the correct root cause is a clear indicator of the HMMs performance as being more desirable than the ANN, the generation of larger numbers of leads is perhaps not. This is seen most readily when faults are not correctly positioned within the list via the HMM.

Direct observation revealed that when this happens the HMM has often weighed multiple correct root causes together. These are features often within the same WMI class related to some dependency or service associated with the fault. When confidence values are the same, the list uses a second-order sort by feature name, alphabetically. Weighing two or more fault hypothesises identically but forcing the correct feature artificially down the list then impacts precision and related metrics. This is a topic sometimes addressed through feature locality [31], a topic discussed further in the following subsection.

For example, when the IIS web services are disabled using the ACC approach, four properties are returned that are all correctly associated with the change. The correct lead is labelled with an *S* (*i.e.* state), but because these properties are equally weighted and some are labelled with

values closer to *A* then *S*, it pushes the correct lead further down the list.

One improvement might be to associate the root attributes – the data listed in the WMI path between the root classes and end features – of a lead and then list properties in a hierarchy. This would allow the root cause to be diagnosed as a subset of a specific feature and may lead to greater precision in future approaches. Unfortunately this is something that wasn't considered at inception. For consistently, this approach is kept throughout.

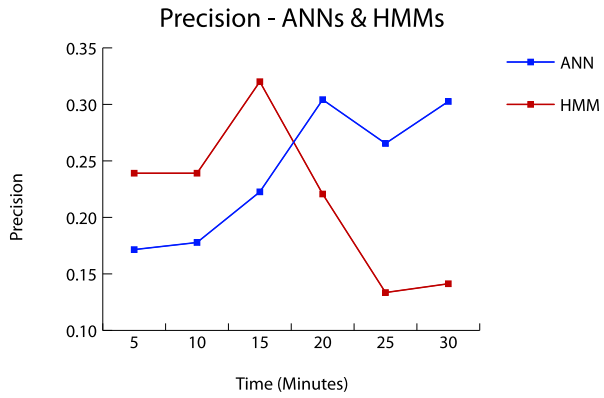

Figure 5.5: FDF v1.0 - Precision. HMMs provide more precise results initially, but eventually trade with ANNs. This is significant in that neither approach is particularly precise, but as more information is added, the HMM appears to drop in precision. This result correlates with more leads being generated, and some second ordering issues.

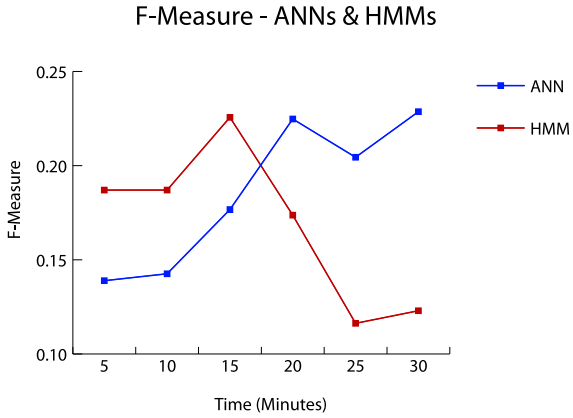

Figure 5.6: FDF v1.0 - F-Measure. The F-Measure represents the overall performance of the learning algorithms in relation to both Baum-Welch (HMM), and Naïve Bayes (ANN) in terms of precision whilst accounting for outliers. Due to the way the trials were executed, similar results were obtained in each examination.

Precision describes the relative position of the correct leads within the ordered list with respect to the total leads generated by the FDF (Figure 5.5). Having only leads that are correctly associated with the fault shows higher precision, as does having localised groups of correct leads. To achieve the latter, a measurement by hand is performed since evaluation of the same feature instance is not supported natively in the FDFs.

The HMM categorises most leads with a greater degree of precision than the ANN. However, as the number of samples increases this position reverses itself and the ANN shows greater precision than the HMM.

The expected result is that the number of root causes suspected by each respective approach increases in number with the number of total samples. Based on the observed data it would appear that this metric actually peaks at about half of the total number of samples. The

causal factor as to why this occurs remains unconfirmed, but it is suspected to be related to the aforementioned second ordering problem combined with increased list size.

Another theory is that infrequent changes in features essentially start an examination process. This means that although there may be 30 samples of data collected overall, perhaps only 5 exist for a specific feature. Without smaller amounts of information features are more difficult predict – a problem that is exacerbated if information is expired prematurely due to a small maximum configuration sample size.

It would be worth exploring a larger sample size of systems' configurations to see if either FDFs' precision values continue to peak at 15 samples, or if there is a trend towards $\frac{1}{2}x$, where x is the number of samples the FDF has access to when the fault is detected.

To test whether or not the value of 15 has a special importance, the window size would need to be modified such that the median was substantially larger or smaller than 15. Doing this would necessitate an entirely new set of experiments that would not be directly comparable with these results – hence why they were not immediately explored. However, exploring this property is looking to be more necessary to optimally explore the capabilities of RBMs.

The f-measure, which in this instance is represented similarly to the precision metric, takes into account outliers and subsets of the sampled data by examining the number of relevant leads versus relevant leads detected by the FDF (Figure 5.6). The results are similar between both the f-measure and precision as there are few outliers. In fact, there is only one series of trials in which a correct fault hypothesis is not generated by an FDF. However, even in that instance the fault is detected.

The performance of the HMM exceeds that of the ANN in many of the trials performed. The most notable exception is seen in stability of ordering and generating fault hypothesises. Although results appear deterministic, between trials the fault indexes of root causes can shift. – the contributing factors of which are previously discussed.

The second FDF experiment attempts to build on the baseline established in the prior experiment and demonstrate improvements using as many similar conditions as possible. This includes volume of data, polling interval, feature selection, and numerous other properties.

In summary, using an RBM to detect faults comes with the costs of higher time-taken, noticeable variability within the fault index results, and, ideally, larger training sets but ultimately demonstrates some positive results. Although it does not outperform the previous experiment in all cases, it does in later trials and it shows potential for more advanced behaviours.

In the RBM trials, the number of training sets is identical to the number used in the previous experiment. However, this volume of data represents only half of the ideal amount needed by the RBM. This is because full training of the RBM normally cannot start until the maximum sample size is reached. This is due to a restriction on the dimensions of the input data. Despite this, results are competitive in a number of cases.

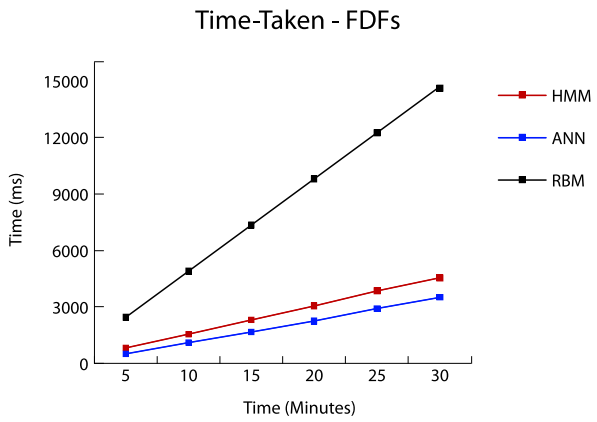

Figure 5.7: FDF v2.0 - Time Taken. A switch from greedy to lazy data ingest caused an expected increased time-based performance metrics. This is because all of the calculations for training the RBMs was performed once a fault was detected versus after every collection sample.

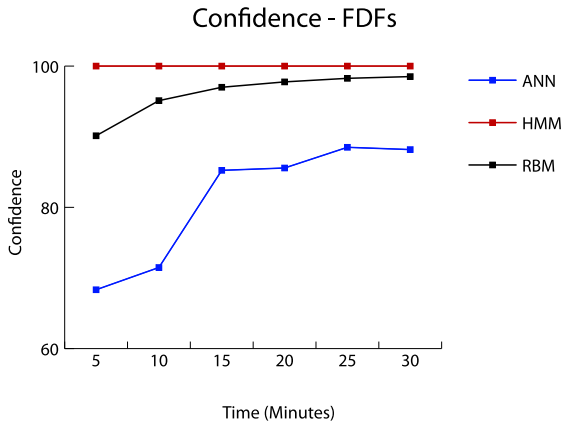

Figure 5.8: FDF v2.0 - Confidence. The gradual increase in confidence values via the CDL appears to be more robust than previous approaches. Rather than a relative value being given or steep increases, a gradual pattern emerges as more data is fed into the FDF.

The RBM requires more time to complete its training and evaluation tasks than the prior approach (Figure 5.7). This is expected to be due to the fact that this version of the FDF does not pre-compute the vectors before a fault is detected. Additionally, the increase in time per minutes of sampled data appears to be associated with the number of RBMs generated during fault identification. The more primitives generated, the more time is necessary to examine potential leads. Performing all of the calculations at a single time provides an advantage, however, in that overhead during the operation of the RBM-based FDF is heavily reduced.

Confidence is markedly improved under the RBM conditions (Figure: 5.8). By comparing the evidence between this [19], previous [18], and other experiments [28], improvements are believed to be due to the feedback mechanism used when training the primitive. Rather than returning a relative value, gradual reinforcement is illustrated as more data is fed into the primitive. This is both a desired and expected result, and one that appears to be produced with more predictability than in the previous approach.

There exists some variance in both the time-taken and confidence metrics. Using the resources

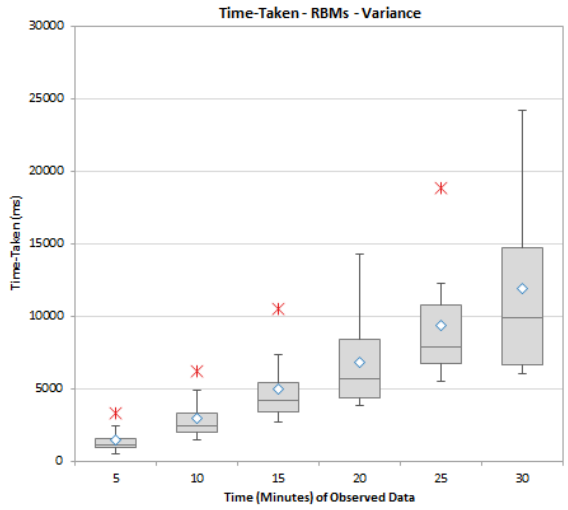

Figure 5.9: FDF v2.0 - Time Taken - RBMs - Variance. The computational time to generate leads using RBMs remained largely predictable. The greater the number of features identified for initial investigation, the more time required to complete the calculations. Variance occurred due to the number of leads needing to be investigated. Note: HMM and ANN trials are not included. See Page 105.

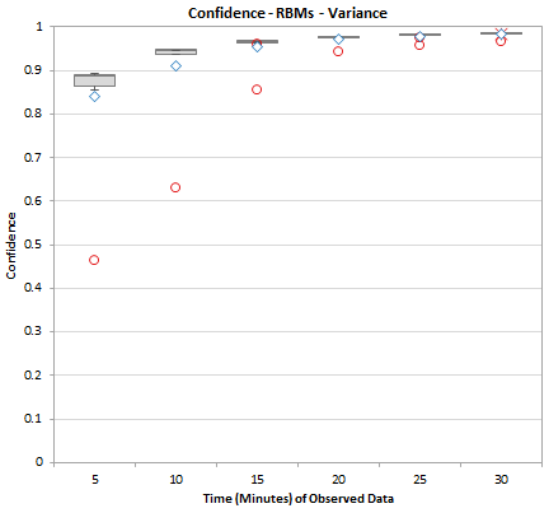

Figure 5.10: FDF v2.0 - Confidence - RBMs - Variance. Confidence values generally display little variance. The learning rate of CDL operated as predicted in nearly all cases with one notable exception where the number of features produced by the fault injecting were smaller than previous trials. Investigation into the cause of this yielded inconclusive results.

provided to the VM it takes an average of about 12 seconds to parse all 30 samples in each iteration. This value can vary depending on the volume of data collected, and how many leads being investigated. Each lead will require a new RBM to be generated. As more data is added a wider range of variance is observed – and in some cases, certain tests produce more variance in results (Figure 5.9). In the latter case, Red, $*$-like markers indicate a near outlier, whilst Red *o* symbols indicate a far outlier. These are observed values that were 1.5 or 3 times the median distance between the upper and lower quartiles, respectively.

Near outliers for time-taken are heavily associated with one specific test – those related to causing faults in Window's disk services. A larger number of changes were produced in the faulty configuration which in turn increased the number of RBMs needing to be trained. In two cases this value is just inside the delimiter between an outlier and standard range data. This same test had similar results for confidence, fault-position, and accuracy which can also be seen in subsequent results.

An opposite trend in variance is observed in the confidence values as seen in time-taken (Figure 5.10). With more data comes less variance, and seemingly more accurate predictions (Figures 5.14, 5.16). This observation comes from cross referencing the variance in confidence with the

average fault index (Figure 5.11), and the variance in the fault indices (Figure 5.13).

In the majority of cases the RBM is able to consistently produce a lower index for the corret root cause than previous approaches (Figure 5.11). When using 30 configuration samples the HMM is able to identify a correct feature slightly more accurately than the RBM with an average fault position of 0.83 versus 0.838, respectively. The gradient of each of these approaches suggests that the HMM could continue to outpace the RBM and further research is required before drawing stronger conclusions.

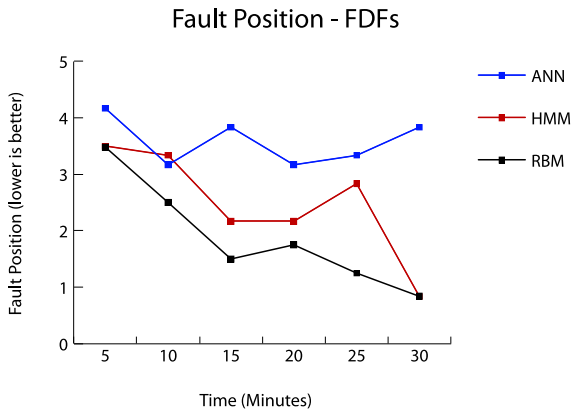

Figure 5.11: FDF v2.0 - Fault Position. The average fault position when using the RBM appears to be competitive and often outperform other approaches.

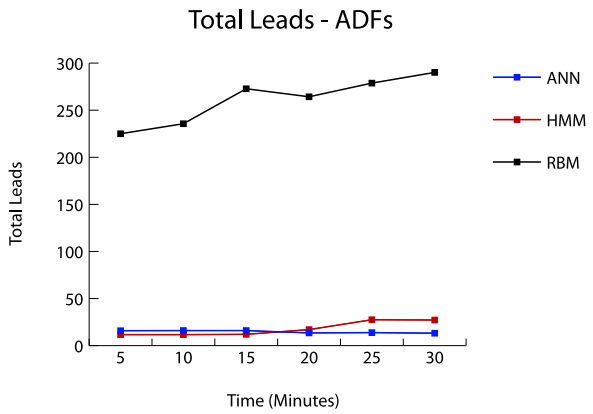

Figure 5.12: FDF v2.0 - Total Leads. The total avenues for investigation are much higher when a lazy data ingest is used.

The RBM produced a higher number of leads than the two previous approaches (Figure 5.12). By switching from greedy to lazy evaluation, changes in the system's configuration are accounted for all at once when a fault is detected. This poses challenges in terms of discerning which leads are more or less correct. The result seems to be near linear growth over time for the total number of features that need to be evaluated. These values provide supporting evidence for the increases in the time-taken data (Figures 5.7, 5.9, 5.22).

In the original series of trials there were often more than 250 total leads. However, during a separate trial for measuring variance fewer total leads were returned. An investigation into the cause revealed deprecated functionality of WMI and migration of certain APIs caused by installing security patches between trials[1]. It showed that the same number of features were being sampled, but a higher than expected number of 0 values were being returned.

Interestingly, total leads does not seem to influence the accuracy of the RBM negatively (Figures

[1] https://technet.microsoft.com/en-us/library/Hh831568.aspx

5.14, 5.16). Based on direct evaluations of the average correct fault position, a correct lead is given more precisely and more accurately than previous approaches.

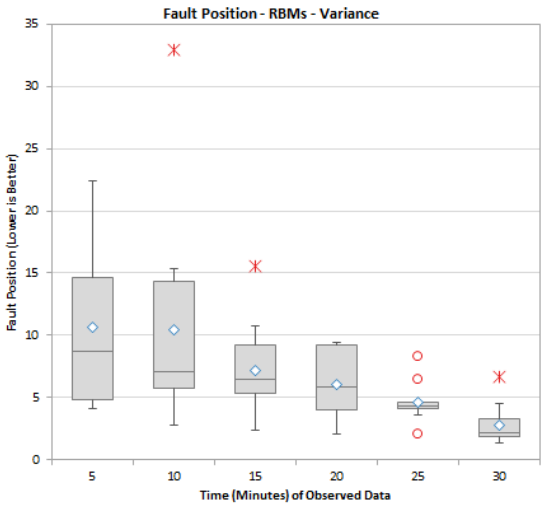

Figure 5.13: FDF v2.0 - Fault Position - RBMs - Variance. This graphs shows results of an identical feature being selected between trials. Normally, the first related feature to the root cause is selected from the list by a test administrator. In this case, only the original feature is selected to help illustrate variance. This helps provide an understanding of how the RBM makes suggestions. At 20 minutes all correct features are categorised within the top 10 leads.

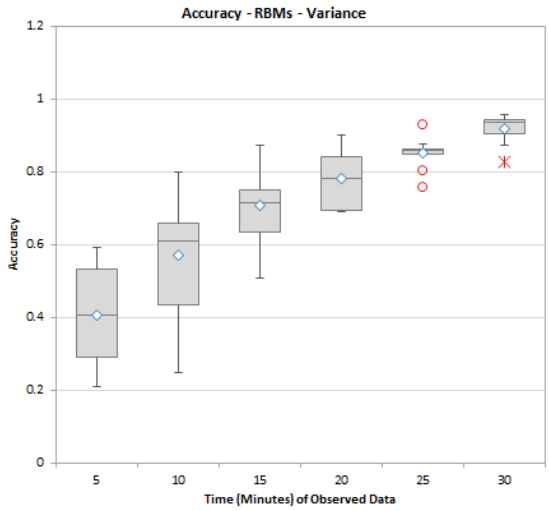

Figure 5.14: FDF v2.0 - Accuracy - RBMs - Variance. The accuracy of the RBM approach increases whilst variance decreases with the exception of differences between 25 and 30 minutes. The reason for this is not fully confirmed. An initial investigation showed maintenance tasks being executed by the operating system at regular intervals set for every 30 minutes. This may create additional features for investigation and account for differences in the 30 minute trials.

Variance in the RBM's fault position output seems to be at least partly associated with how the RBMs are instantiated (Figure 5.13). A random seed is used to help build weights and biases within each neuron of the RBM. This value dictates the initial state of the neuron, and consequently the paths for each output are somewhat different if these are adjusted. Since the object are constructed upon fault detection, different RBMs exist between different trials. This seems the most likely explanation for the higher rates of non-deterministic output seen in previous trials.

Both the fault position and accuracy values show strong results compared to those in the previous experiment. A majority of fault indices appear to be lower, particularly after the 20 minute mark, as well as the previously described increase in accuracy. Still, there is room for improvement.

As mentioned, the ideal FDF will return a list of fault hypotheses that consists of only correctly

associated leads to the detected fault. For continuity, this experiment continues operate on the assumption there is a single feature being returned indicating the absolute root cause of a fault. This continues to be a less than ideal measure as it is often the case that multiple features can be correctly abstracted as part of the diagnosis. With the randomisation of the RBMs, however, some variance occurs in which feature is placed making this an arguably more noticeable phenomenon in slight increases in fault index (Figure 5.15). It seems that accuracy is improved over other approaches despite this (Figure 5.16) with fewer false positives being present overall in the results.

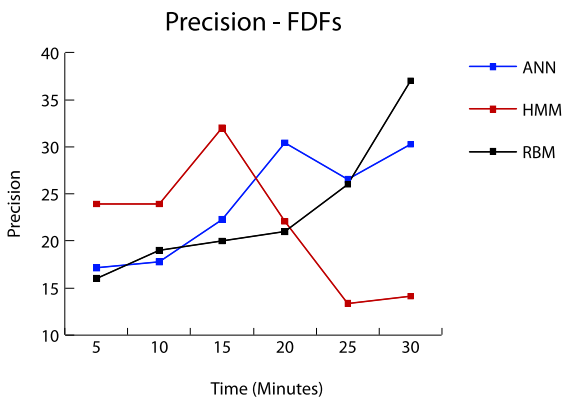

Figure 5.15: FDF v2.0 - Precision. Precision of the RBMs show a marked improvement on previous experiments, however some ramp up time is necessary. Averages of results show a marked improvement between the 20 and 25 minute marks and a stronger trend than prior approaches.

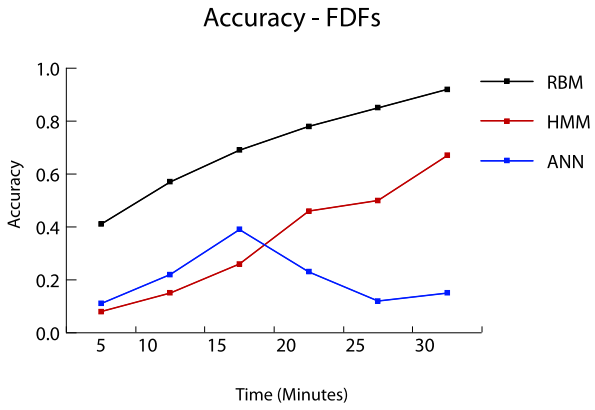

Figure 5.16: FDF v2.0 - Accuracy. RBMs show an improvement in accuracy over other primitives. Initial values start out low, but higher than other approaches. They continue to show steady improvement with the possibility of meeting the accuracy of the HMM.

A couple of outliers are present in the results and this is directly observable in the variance graphs for precision and f-measure (Figures 5.17, 5.18). In trials where fewer numbers of leads were returned, precision seemed to increase. The fact that the correct root causes are still listed, might imply that leads are sometimes harder to rule out than they are to confirm as fault sources. However, this claim would need further testing to be substantiated.

The overall range of values produced in the trials for the RBMs modest improvements. There are a couple of tests where it took longer for the precision metrics to increase than expected. Direct observation of those tests revealed ambiguity in the results of certain experiments, where a feature or series of features appeared to be missing from the data collection tables. This seems to imply a problem exists at the data gathering stage via WMI versus the RBM's learning capabilities.

A sharp increase in desirable performance metrics and observed traits in multiple graphs after 20 samples. A longer running test seems to be necessary to understand the limitations of the experiments that have been performed. Unfortunately, it was assumed that 30 minutes would be sufficient, where as it seems it is perhaps not.

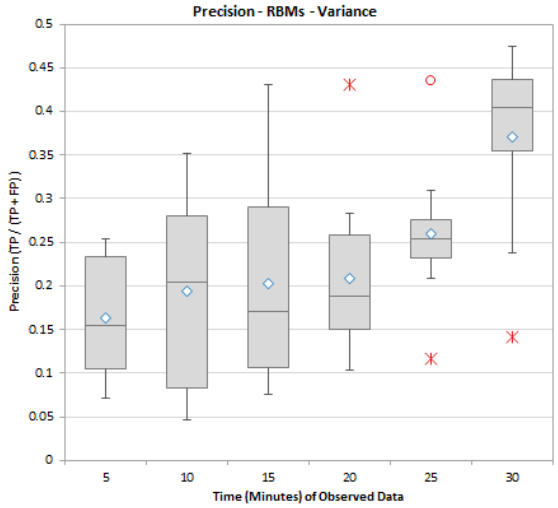

Figure 5.17: FDF v2.0 - Precision - RBMs - Variance. Precision tends to increase as more data samples are used. This coincides with previous observations at around 20 minutes where greater increases start to take hold.

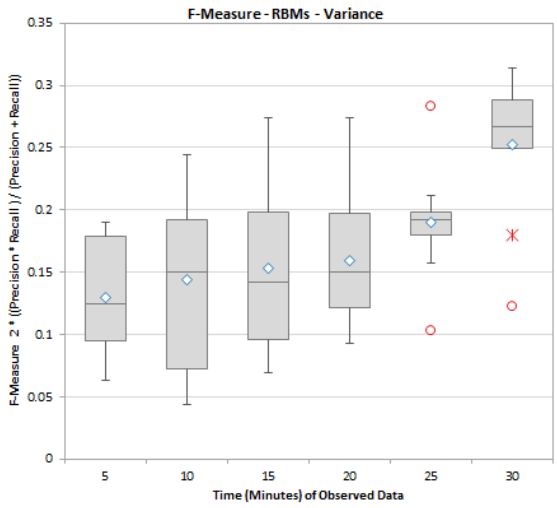

Figure 5.18: FDF v2.0 - F-Measure - RBMs - Variance. A number of minor outliers occurred during the course of these experiments. Some tests provided a larger number of features to examine than were expected, and variance in fault index had an impact on results.

5.3.2 UBL

UBL's results show that a SOM can be used in some situations to forecast feature behaviours accurately far enough into the future to take corrective action before a fault fully manifests – notably when the fault manifests slowly in non-noisy datasets. However, noise and limitations on the amount of data that can be observed before convergence and a forced re-instantiation make this approach arguably less effective than alternative approaches. Evaluation of UBL is based upon a comparison between the results via the SOM and two unsupervised learning schemes: PCA and κ-NN.

Detailed results from the UBL experiment can be found in the original publication [28], but relevant results are summarised here for convenience. The majority of results associated with the UBL approach are presented in graph form using the aforementioned ROC curves and their associated bar charts. These graphs describe the fault prediction accuracy of UBL via

true positive rate and *false positive rate* performance metrics using the SOM, PCA, and κ-NN (Figure 5.19).

$$A_T = \frac{N_{tp}}{N_{tp} + N_{fn}} \tag{5.1}$$

$$A_F = \frac{N_{fp}}{N_{fp} + N_{tn}} \tag{5.2}$$

Figure 5.19: Prediction Accuracy Formulas: UBL

4.1: *True Positive Rate*
4.2: *False Positive Rate*

Furthermore, UBL consistently provides two sets of results for each ROC curve – a non-smoothed series and 5 point moving average smoothed series. These datasets are labelled as UBL-NS and UBL-5PtS, respectively. The graphs demonstrate supporting evidence for several advancements in unsupervised fault detection as evident by suite and application. These results are described in terms of *accuracy*, *lead time*, and the effects smoothing has on fault identification.

In low noise datasets (*MemLeak*, *CPULeak*) with *RUBiS*, UBL claims a high rate of true positives – up to 97%. This is contrasted with the slightly noisier dataset produced by NetHog with an 87% true positive rate. Notably, the higher accuracy in the former is attributed by the authors as being due to the gradual nature of the fault's instantiation. Faster occurring faults are claimed to be less likely to be detected under all three algorithms.

Smoothing in the *MemLeak* is also attributed as being beneficial to the true positive rates in the test – a claim of up to 20% greater true positive rates than without. When the dataset's results are modified in this way, they are more likely to remove outliers associated with what the authors describe as "*transient noise*". The gradual manifestation of the fault then supports a higher true positive rate since it is less likely that data associated with the correct fault hypothesis is removed.

Test results using IBM's *System S* return the most successful results. A true positive rate of 98% is achieved on average, with individual tests such as *CPUHog* achieving 93%. False positive rates for these tests are reported to be as low as 0.5%. High true positive rates are again associated with their longer instantiation times, including another testing tool called *Bottleneck*

which manifests quickly producing lower success rates. A skew in the dataset is noted in that less noise is associated with how *System S* returns its sensor data.

It is here that smoothing is described as not being particularly helpful and that the associated benefit from its inclusion is directly related to the volume of noise involved. That is to say, when the observed information frequently peaks and troughs between minimum and maximum values it is less likely to be predicted correctly – a problem that is possibly exacerbated when you normalise values. Additionally, as smoothing can remove critical points of data, it sometimes helps to increase the false positive rate – as seen in this instance. Regardless, the best results for *Bottleneck* are when smoothing is not used.

The *Hadoop* results are divided in terms of the best true positive rates between *MemLeak* and *CPUHog*. As both of these tests cause faults to instantiate rapidly, and since the *Hadoop* datasets are the noisiest, results from these tests show the highest false positive rates. Again, smoothing is attributed to the removal of some critical points of inference, as noted when the authors use both 5 and 50 point smoothing to try and achieve better results under this specific set of tests. Unfortunately, true and false positive rates are not directly provided in the text, and instead are inferred through manual calculation (as described in Section 5.4).

The most important results from UBL are arguably the *lead times* it produces. One critical difference between UBL and the FDFs is that the former focuses on a proactive solution to anomalies. To successfully complete self-healing operations, UBL must be able to recognise a pre-fault state and recommend a recovery strategy before the fault fully manifests.

UBL claims an average of 38 to 40 seconds of *lead time* for the *CPULeak* test under RUBiS; for *Memleak*, the results average only 7 seconds. This variation in results is explained due to variation in the 'background noise system' influencing the input data, but arguably such circumstances can be expected under normal operating conditions. Similar results are observed under the *System S* tests – 47 seconds of lead time, on average, for *MemLeak*, but as few as 3 seconds for *CPUHog* and 5 for *Bottleneck*. *Hadoop* results are similar still, with 24 seconds of *lead time* for *MemLeak*, and 3 for *CPUHog*, respectively.

A summary of the lead times generated by UBL is provided for ease of reference: (Table 5.1)

How much time is necessary to correct a fault is an open question. Supporting work for UBL claims a range of 10 to 30 seconds [104]. Objectively, this means that under non-noisy datasets where faults manifest slowly, the UBL approach is effective at generating a correct fault hypothesis.

	Hadoop		System S		RUBiS	
Testing Suite	Avg	Max	Avg	Max	Avg	Max
Bottleneck			5	6		
CpuHog	3	4	3	4		
CpuLeak						40
MemLeak	24	25	47	50	7	50
NetHog					7	7

Table 5.1: Lead Times: UBL. This chart represents the number of seconds UBL identified a failure before it reached a terminal threshold; higher values are better. Blank = No Data.

5.4 Discussion

Differences in the approaches of both the UBL and the FDFs make direct comparison difficult. This is specifically due to the lack of common tests between the UBL testing suites (see Table 4.3), and the lack of detailed, publicly accessible results. To mitigate this issue a common baseline between each approach is synthesised using similar performance metrics and criteria whilst also acknowledging their fundamental distinctions.

Data was extracted directly from the original publication via XML contained within the ROC graphics. The ROC curves describe UBL's results in terms of "fault prediction accuracy" via the true positive rate and false positive rate metrics (Figure 5.19). . Using this information, a common baseline between each approach is synthesised by extrapolating similar performance metrics from the raw data in the results from each experiment. All associated data is made public (Appendix A.1).

There are a number of differences between UBL and the FDF approaches both in terms of the data provided and their fundamental behaviours. In addition to sampling frequency, major distinctions include: The type and volume of data being sampled, forecasting capabilities, proactive versus reactive behaviours, and classification criteria. The resultant data for each of these studies is thus presented differently.

Using the publicly provided FDF results for *fault position*, *time taken*, *precision*, *confidence*, and *total leads*, and the various true and false positive rates from UBL the following performance metrics are synthesised: *precision*, *prediction time*, and *fault position*. Further metrics are not generated as some of their fundamental values are missing from UBL's public results and were not obtainable upon enquiry.

Precision data is generated by taking true positive rates from UBL and the FDFs and averaging the values of all relevant experiments at identical time intervals.

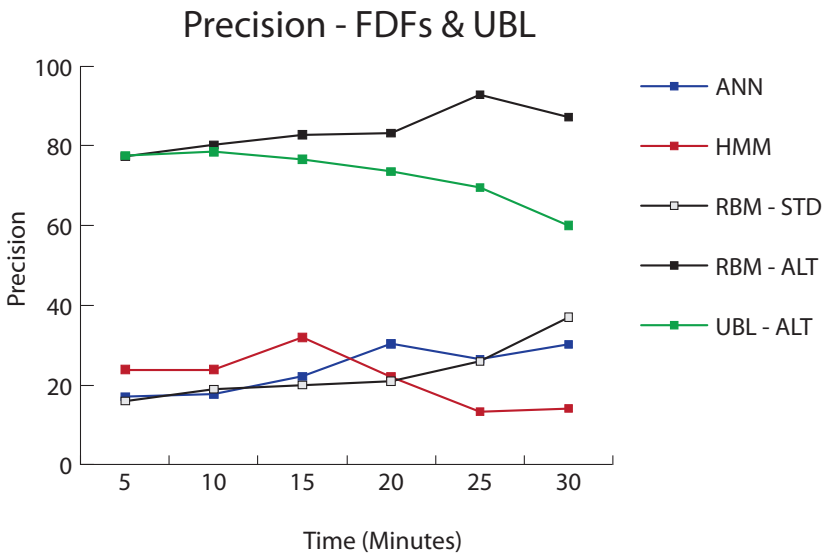

Figure 5.20: Precision Measurements: UBL & the FDFs. The precision of both FDF approaches remains low, however the RBM approach shows a promising trend as more data is added. UBL's precision drops the more data is added. The first three metrics show results for fault identification where all features above the correct root cause are considered false positives. The bottom two results (-ALT) show precision for fault detection.

In the case of the FDFs this means using values for tests that leveraged the same primitives in 5 minute intervals. Each interval represents a sixth of the total results. Similarly, the UBL data uses both the NS and 5-PtS datasets to generate the precision data points at intervals that matched a sixth of the volume of data.

The definition of precision is different between UBL and the FDFs. UBL determines a true positive if the selected feature is both correct and returned within a specific time period:

> ''*We say the models make a true positive prediction if it raises an anomaly alert at time* t_1 *and the anomaly indeed happens at time* t_2-$t_1 < t_2 < t_1 + W$, *where* W *denotes the upper- bound of the anomaly pending time*".

Per their original paper, W is decided arbitrarily by Dean, *et al*, after manual observation of prior results – see page 6 of the original publication [28]. This is a definition that is incompatible with the FDFs for two reasons: It is based on fault detection and not identification, and it emphasises forecasting within a specific time period rather than accuracy in diagnosis.

Because the FDFs retroactively investigate SLO violations, a direct comparison using identical definitions of true positives is not possible. Instead, using the earlier started goal of automating the root cause analysis, we can assume a similar comparison by returning a true positive if the

root cause is correctly identified within the first 10 leads. This condition has been chosen based on an anecdotal assumption that an engineer looking for a root cause would be willing to look through such a list.

For reasons of comparison, Figure 5.20 shows two sets of data. The first three metrics show the traditional definition of *precision*: N_{tp} / $(N_{tp} + N_{fp})$, where a N_{fp} is, as stated, the number of features listed above the correct root cause of a fault – *i.e.* the fault position. The latter two metrics in Figure 5.20 describe when a fault is correctly identified within an arbitrary time bracket (UBL - ALT), and when a fault is correctly identified within the top 10 features once the fault hypotheses are ordered (RBM - ALT).

RBMs show continued improvement overtime when compare to other approaches – a situation that could be explained by possible over-training. As more data is received by the primitives and learned, their sensitivity to new information is reduced, but they do not converge or lose precision as UBL seems to.

A lack of sensitivity is more evident in situations where learned information is not expired. UBL lists a known convergence problem after too many learning updates to the SOM – this effectively limits the maximum operating time of this approach, a problem the authors work-around by using forced reinstantiation and training of the SOM at periodic intervals. Conversely, the FDF experiments use a rolling window of information to make inferences from as specified at run-time by a user. This value is derived via the the polling frequency (in milliseconds) and total number of samples to keep. Although both experiments incur degradation, UBL's appears to be much more rapid – however the variability in the RBM data makes it difficult to be certain in all cases.

Using true positive rates demonstrates that if the correct solution is discovered, in many instances multiple potential faults are also provided. In UBL, neurons 'vote' between possible root causes, whilst the FDFs use confidence values based on prediction likelihood. Both experiments then order their respective fault hypotheses. Arguably, both experiments can be seen as basic recommendation engines with solutions being ordered or weighted in some fashion. By understanding where the correct fault is within these engines – either by weight or by position – a demonstrable type of effectiveness is provided for each approach (Figure 5.21).

The fault position of the correct root cause is evaluated against the total number of recommendations (Figure 5.21). The lower the value, the sooner the engine selects the correct root cause. FDFs using ANNs are the most consistent technique when examined via this performance metric, exclusively, but this level of performance is eventually matched by both RBM and HMM given a sufficient volume of data.

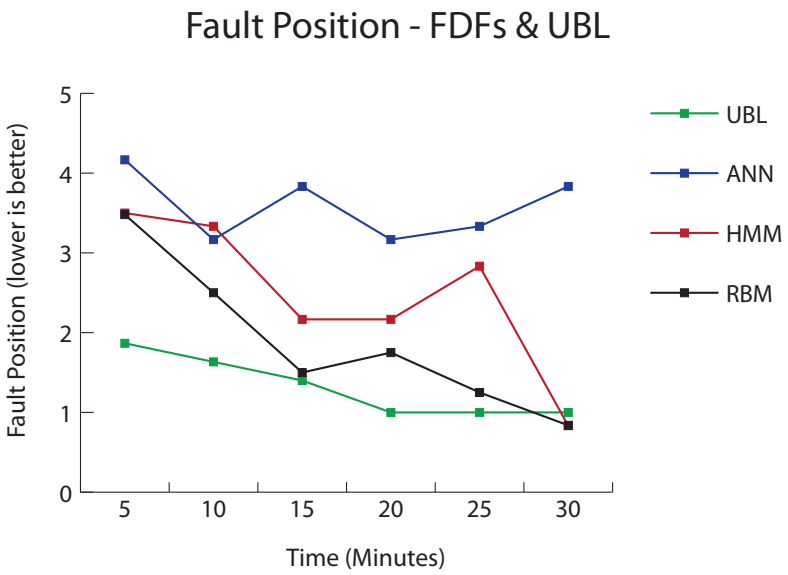

Figure 5.21: Average Position of Faults Based on Approach: UBL & the FDFs. UBL and the FDFs prioritise potential sources of faults. Correct recommendations are represented as an average of all tests based on primitive type. Lower values signify better recommendations.

Accuracy in predictions is often directly related to resource availability. The balance between how fast an application returns a result and its level of accuracy is paramount. Fault position is therefore contrasted with resource utilisation by examining the total amount of time a framework took to indicate the source of a fault – collectively referred to as *prediction time*. Prediction time is based on the total number of milliseconds from when a fault was first suspected and when the results – an ordered or weighted list of fault hypotheses – are fully produced by the primitive(s).

In instances where greedy algorithms are used the amount of time it takes to predict a fault is fairly static. This is an expected result as the systems in question process the same amount of data in the same fashion at regular intervals. However, the FDF using a lazy implementation with RBMs shows a varying amount of time to process information (Figure 5.22).

UBL reportedly takes a static 490ms per minute of data gathered to update the SOM before generating a prediction. Using this value the total amount of time per sample is plotted out in minutes to match the interval results of the FDFs. The FDFs initially followed the same pattern as UBL – although due to the complexity differences of their respective learning algorithms they execute much faster. The one exception being that CDL requires nearly the same amount of time as the SOM to update its neurons – despite the former's lack of continuous data ingest.

Implementation plays a role in the evaluation of time-based performance metrics. The two

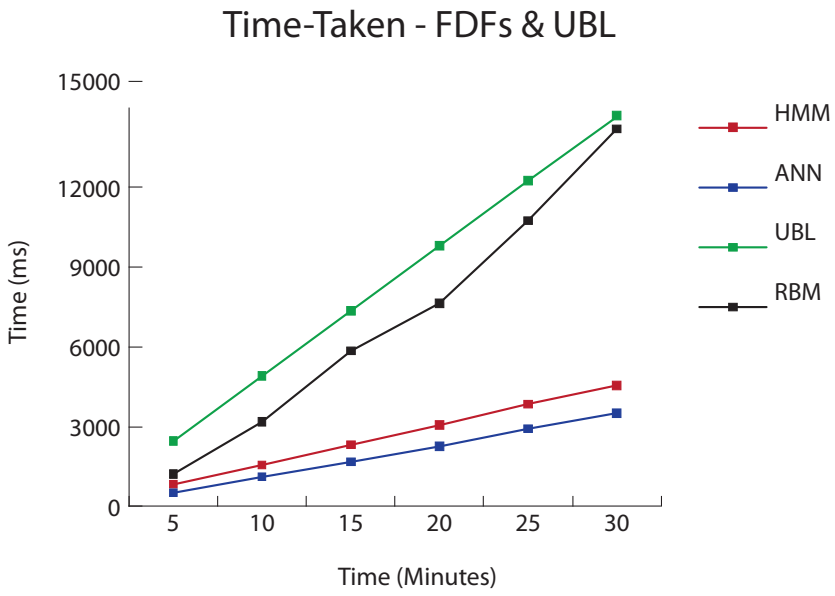

Figure 5.22: Time Taken Performance Metrics: UBL & the FDFs. The UBL experiment does not post timing data but instead reports performance as a function of total samples. Additionally, training times are also reported to last until each neuron has been updated 10 times making variance a possibility. This information is not given in the original study by Dean, et al [1]. The FDF experiments do provide timing data with greater variances being confirmed when switching from a greedy to a lazy ingest.

primitives in the first FDF experiment leveraged a greedy implementation using a windowed collection of datasets. This means that once a minute all primitives are discarded and retrained – an action requiring 50% of the total CPU activity on the VM for about 15 seconds. By processing this data upfront the system is able to return results relatively quickly – between 500ms and 4,500ms depending on the size of the dataset being parsed.

However, the impact to each system's performance is clearly a disadvantage. UBL's linear training and fast prediction times illustrate an effective approach for determining errant feature behaviours within 2,450ms and 14,700ms (Figures 5.1, 5.22) $\pm$ 2.5ms. As UBL's primary goal is to proactively predict anomalies this time is particularly important. Faults that are identified quickly enough could ideally be addressed before fully manifesting. This is a fundamental difference from the FDF approaches which emphasise reactive behaviour by updating future iterations of VMs.

RBMs in the FDF approach compare similarly to UBL time-wise when predicting the root cause of a fault. A substantial increase in time is noted between the two FDFs – the latter, again, based on a lazy implementation for data ingest. This accounts for both the increase and difference between the processing times for the same volume of data in the other FDF experiments. Times range from ~1,250ms to ~14,250ms depending on the number of samples provided to the FDF.

Direct observation shows that prediction times plateau once the maximum number of samples is reached – in this case 30.

Prediction time shares a relationship with training time. If the majority of training comes before the prediction takes place, then the prediction time is reduced. Notably, training time for the primitives varies based on a number of characteristics including required epochs, neurons per primitive, volume of data, type of data, and of course which learning algorithm is being utilised. Of the two instances where lazy algorithms are implemented training times appear to be similar.

Neuron count totals 1,024 and ~7,200 in the SOM and various FDF approaches, respectively. Primitives incorporating these neurons require up to 6,000 training epochs for UBL, and 5,000 per primitive when RBMs are in place. In other FDF instances training periods use as few as 5 epochs per primitive. Naturally, depending upon which learning algorithm is in place, the time for completing training in a primitive varies. Clearly different learning algorithms have different rates of success, but their overall effectiveness is also bounded by resource constraints. Given a greater number of resources – such as memory and clock cycles – the accuracy of the predictions increases, but only to a point. However, in the vast majority of the examined cases it is possible to generate and select an accurate fault hypothesis using stochastic primitives.

A shift from reactive to predictive measures is currently underway within unsupervised fault detection for self-healing systems. Specific attributes can be correctly associated with a fault using abnormal variations in either performance metrics or raw frequency analysis of feature changes. However, there is room for improvement in these approaches – particularly in noisy datasets, feature locality, and distributed learning.

Other approaches in the reactive space may yield better results under the ephemeral computational model of cloud computing. Allowing systems to fail has some benefits in resolving deterministic fault loops – the first steps have been taken, between generating an accurate list of potential root causes [7, 18, 19, 20, 28], and the ability to synthesise new, valid systems configurations [27]. Not all results meet with their respective expectations. Due to the volume of data, the FDFs are expected to take longer to find a solution than UBL. Instead, the time values are similar but the accuracy values are not. Two important inferences are gained from this observation.

Firstly, accuracy and resource utilisation seem to share a relationship in the FDF experiments. By increasing the number of training epochs it may be possible to achieve greater accuracy – using both markers could produce a way to cross reference efficiency in future approaches.

Secondly, by comparing the precision (Figure 5.20) and fault position (Figure 5.21) per-

formance metrics, it is evident that the RBM approach demonstrates fewer false positives than UBL. This is excellent news for environments seeking to reduce type I errors, but the computational costs of using RBMs could benefit from optimisation. Additionally, other types of primitives may yield stronger results – a topic for further exploration.

Training periods are necessary for both the FDF and UBL approaches before operating. Whilst an improvement over prior research in their ability to use unlabelled data, the existence of these requirements represent a fundamental problem: How to balance instantiating a framework that can accurately detect faults and reducing the initial training period. Several problems have emerged in trying to balance these two factors. However, there may be a solution to this problem using an evolutionary approach.

SLOs and performance tests provide measures of a system's health, whilst offering a way to administer a system from a higher administrative level. This is one of the primary goals of self-managing and self-adaptive systems research [2, 1]. In each of the aforementioned studies, progress in this area is apparent. However, not all SLOs are created equal. During normal use variance needs to be accounted for whilst preserving contextual validity. The FDFs accomplish this using a windowed approach, but this approach is notably absent in UBL.

In each experiment the use of presumed or verified datasets help classify sampled information. The FDFs' use of performance tests to determine the general health for the system allows for faster training, but does not take into account individual feature changes until after a fault is detected. This makes feature locality more difficult to determine in its current form – a critical step in subsequent phases of research.

Similarly, UBL uses a vetted series of inputs to resolve a number of factors associated with SOM instantiation and training. However, it requires several stop-gap procedures to operate such as the periodic re-instantiation and retraining of the SOM. Periodically interrupting service availability is unacceptable in practical implementations, and rebuilding the SOM comes with a long training delay effectively causing built-in outages to UBL's self-healing capabilities. Additionally, the use of a static training mechanism is, arguably, a potential source of problems for dynamic fault detection, *a priori*.

The use of a layered approach towards classification is one of the primary differences between the FDFs and UBL. The UBL study ignores so-called 'constant' metrics – those values that change very infrequently – in favour of minimising resource usage. However, the FDF approaches do exactly the opposite: They reinforce non-changing attributes as nominal behaviours and use this information to update confidence values when changes occur and only when necessary. This is an important distinction at multiple levels – noise can be normal, a fact

entirely ignored by UBL. By attempting to mitigate such factors, much of the information is irreparably lost, and arguably, more likely to induce errors.

Conversely, not having to write independent policies is one advantage that UBL has over other approaches: No policies need to be written explicitly outlining the purpose of the machine. Normalising all values and providing a static minimum and maximum allows for autonomous evaluation of the system regardless of role. This means faster provisioning but, again, less targeted behavioural adjustments after instantiation.

UBL also analyses some self-healing aspects that are beyond the scope of the FDFs – such as comparing centralised versus localised training of primitives by exporting information to a training VM. The goal of delegating the training of the primitive is to try and centralise the re-population of the SOM, but doing so requires all training and corrective actions to complete before said fault fully manifests. This needs to be accomplished within the lead time generated by UBL, and, unfortunately, the time to transfer data to and from a training VM plus its subsequent implementation often exceeds the lead time that is generated by the SOM.

Acquiring information regarding UBL was not easy. Without being able to instantiate the experiment locally, much work went into dissecting and understanding the exact operation of the system as reported publicly. Inconsistencies occasionally showed up in this exploration, and the result was a limited dataset for comparison. As there were few other experiments at the time to compare against, this left few options for validating the approach. Conversely, this particular style of observation and the use of stochastic primitives in this fashion are gaining popularity.

Chapter Six

Conclusion

This chapter provides a summary of the lessons learned, future research, and conclusions from the aforementioned experiments. In brief, the use of stochastic primitives such as ANNs, HMMs, and RBMs provide valid, accurate approaches for generating fault hypotheses but there is room for improvement in a number of areas.

6.1 Findings

This thesis provides and meets several major claims. The first and most relevant of these is that by building an application that uses a combination of unsupervised learning, stochastic primitives, and performance tests, the root cause of a fault within virtual machines can accurately be identified by comparing a system's observed and predicted feature behaviours. A root cause of a fault can be heuristically obtained by generating values that represent the likelihood of observed changes and cross referenced them in configuration samples that have passed their respective SLOs.

This has been shown in a number of the results, but primarily in the fault index figures. Reducing the number of potential leads to less than 10 occurred frequently, but with optimal results usually occurring after 20 minutes of testing. Variance in output has been accounted for and although decreases in variance occur overtime, returning the correct fault position more consistently is an area for improvement – particularly in trials using less than 20 minutes of observed data. Beyond 20 minutes, an improvement between approaches is demonstrated.

A baseline has been established for the performance of three different primitives using two similar approaches. This has allowed for some comparison between performance of implementations, and provided minor insight into an expectation for evaluating performance against human subjects tests.

The implementation of various self-healing systems has been discussed based on computing environment, learning algorithm, and management style. Although no concrete results can been drawn it appears that contextual use does play a part in the development of self-healing approaches and methodologies. Terminology in this area has also been briefly explored; despite attempts by others to do this already [9], it could use a refresh, This is particularly true for self-configuring versus self-provisioning systems.

Some automation in fault identification has been demonstrated using unsupervised learning. It remains to be seen whether or not an approach like this will reduce costs, however autonomy over supervised approaches appears to be at least partially demonstrated. The identification of the root cause occurs without ever having seen a prior case which is further toward the definition of autonomous systems management [4, 105]. Additionally, the use of performance tests successfully emulated high level management of systems via SLOs. This approach was rudimentary, however, and an engine that generates these tests into code could provide a useful measure of automated in the future.

Differences between application crashes and controlled stops were analysed with results showing clear differences in impact to fault identification in some circumstances. Often, it appeared that crashes were marginally easier to identify if detected within 60 seconds of manifestation. Understanding the full impact of this observation remains unresolved, however.

6.2 Lessons

These achievements did not come trivially. There are a number of hurdles that have been overcome during these experiments. This includes the in-depth examination of related studies, discovering and overcoming technical limitations and constraints within a number of operating systems and environments, and superficially exploring multiple disciplines that border and relate to the study of self-healing systems. In addition to the scientific research gained from these experiments, a certain level of understanding in processes and form have been ascertained along with a number of lessons.

Root causes may be detected differently based on how faults are instantiated. In the

initial outline of this thesis, it was assumed that faults stemming from the same root cause but instantiated differently may produce different results. This thesis explored this idea by injecting two different types of faults: ACCs, and DFIs. The latter is similar to the traditional approach of other experiments, with the exception of how quickly they are instantiated, whilst the former is more akin to user error.

The combination of these two properties made for interesting results. It's clear that user errors sometimes produce more of a footprint to gather information from. Preliminary data shows that self-healing systems are more likely to detect ACCs type issues, but the accuracy of determining the root cause is lower. This is a finding believed to be unique to this study.

The speed at which the faults manifest in this study makes their detection harder to mitigate than other approaches. Although no recovery strategy is explicitly implemented by the FDFs, the detection capabilities need to be more precise than other approaches. Whereas other self-healing systems gradually evaluate behavioural and performance data against hard SLOs, the FDFs need to dynamically adjust their expectations and capture a result in much shorter time spans. Comparatively, tests between other systems would run for 30 times longer than those run in the experiments mentioned in this thesis – and in fewer iterations.

The nearest metrics seen to the FDFs in this respect came from UBL. They ran their experiment "30 to 40 times" for each of 6 tests – where as this experiment ran 30 times for each of 6 tests for the aforementioned two types of faults, 3 times. In total about 10,800 tests have been run so far in this respect and catalogued.

Performance tests with stochastic primitives works as a preliminary policy-engine. The ability to easily specify boundaries for desirable behaviour in a system is not something that has been accomplished universally. In order to interlock with the planned future adoption of Evolutionary Programming techniques, this experiment uniquely chose to implement performance tests as used in GAs.

This has come out with a successful result in terms of being able to specify utility-policies at a high-level to the system and have them understood by the FDF frameworks, however it could be improved. Due to resource constraints hard limits on development of the policy-engine had to be put in place. There are improvements that can and should happen in this area so that specialist knowledge isn't required to continue to develop operational "fitness" policies, and so that the tests are extensible and more easily reusable between systems.

Stochastic primitives can be used to accurately identify potential root causes of faults but not immediately. This has been tested under multiple conditions – including primitive type

(ANNs, HMMs, RBMs, SOMs), learning algorithm (Naïve Bayes, Baum-Welch, CDL), and volume of information (between 5 and 30 samples). In addition, preliminary work has been done to extend these studies for using GSNs – a new type of stochastic primitive that revisits back-propagation style learning – and multi-point forecasting.

Findings so far indicate improvements after 20 minutes, but it remains untested if the number of epochs could be increased to improve this sooner or if another primitive might be more successful than the RBM. Additionally, updated learning algorithms are not readily available and this area remains largely unexplored.

***Drift* between systems' configurations can impact results, and it is hard to minimise.** Understanding how to resume tests using the same information, run these tests in parallel without impacting a hypervisor, accounting for the amount of time between tests, and reinitialising variables precisely in a way that excludes chances for accidental faults in configuration is a non-trivial task. This did not occur perfectly during the initial tests and sometimes they had to be redone in batches – a frustrating but educational experience.

In the end, a schema had to be developed along with a reading, parsing, and strongly typed verification mechanism for storing to and from the file system. This was combined with careful instantiation of VMs on a hypervisor to use separate but identical hardware via system snapshots whilst capturing data by hand – a requirement to validate and look for false negatives. By definition, the FDFs are incapable of understanding when a fault has not been correctly detected.

Using non-simulated data provides an edge and keeps results relevant. In at least one other experiment outmoded data was used to test self-healing behaviours [28]. This was something of a motivation factor in calling out unfit tests, and galvanised the already made decision to use live information for evaluating the FDFs.

Although no samples were provided by a third party or agency, queries via common tools such as cURL and HPING3 were leveraged to generate input for the systems under test. This at least kept the protocols within the correct decade and provided a reasonable level of certainty to the results as these are standard tools of the trade – at least for the moment.

Self-healing systems continue to specialise based on contextual usage and future evaluations should take account of this. In the initial survey of self-healing systems frameworks it became apparent that some technologies are more likely to be implemented under certain conditions. This was most notable in areas where risk and trust were factors – such as ownership of a system in ad-hoc management styles and computing environments [86].

In these cases certain priorities may arise in the development and evaluation of a self-healing

system. A more extensive set of evaluation criteria may need to be designed such that systems can be examined under the requirement of specific roles. This would also help to understand and correlate the behaviours of these systems with their respective uses.

6.3 Future Work

Self-healing systems research continues to face a number of open problems with major areas of emphasis including feature locality and the dynamic generation of recovery strategies.

Even in the tests within this thesis that tried to use single points of origin for failures, multiple factors become obvious after the fact. To understand the root cause of a problem, it is clear that a collection of behaviours must be observed and understood of any single feature. Thus, iterating over a singular point – although useful – will not be enough for professional implementations.

Studies in feature locality are progressing but studies exploring links between the relationships of feature behaviour have not been produced using stochastic primitives. One approach might be to generate hypergraphs of associated features based on their likelihood to change within a certain time interval and then watch for behaviours in those subsets. This could conceivably be done using a similar approach as to the one listed here however it is expected to be computationally costly.

Understanding costs in general for stochastic primitives is not an area that has been greatly explored. A large amount of research remains for the base of such approaches – including how learning algorithms operate, and how to minimise problems such as the accumulation of errors when forecasting. These are not issues specific to self-healing systems, but they are dependency points which should be examined.

In a related context, a number of new primitives and optimisations have been produced. This includes optimisations to backprop via GSNs, and exploration into building fully recurrent neural networks. There appears to be a race between stacking (*i.e.* layering) primitives such as RBMs and developing new technologies. The results of this are impossible to predict, but the training costs of fully recurrent primitives appear to be the largest stumbling block. If not solved, it may be the case that the goal of ‘networked learning’ will be achieved through the former approach – something which is already claimed to be achievable via mathematical proof [14].

When this thesis was first started there were no known experiments in using unsupervised learning and stochastic primitives to make predictions based on feature changes to identify

faults. Within two years, three studies were produced including the two primary experiments in this thesis [18, 19, 28] that explored fault identification under these or similar conditions. Additionally new studies using RBMs and feature prediction began to appear in fields outside of self-healing systems – including computer security and psychology – that focused on vulnerability detection [103] and consumer purchasing habits [106], respectively. It's too early to say if the approach is gaining popularity, but a cursory look shows a number of new studies in 2015 have also started to emerge.

As expected, no human subjects tests exist within self-healing systems. The lack of opportunity for these tests – including the collection of suitable subjects, participation from eligible agencies, and the approval of such studies – make for difficulties in achieving this goal. However, progress in this area would allow for definitive responses to questions about effectiveness.

It seems that a divide in research is occurring based on the environments under which self-healing systems operate. As systems become smaller and less integrated into large data centres, reliance is being moved toward the client for certain properties and responsibilities – and for sensor data. Few studies exist on these emerging business models and how they impact decisions on self-managing systems. Basic observations include data integrity problems being mitigated by shifting toward specialised, centralised infrastructures – effectively removing such concerns from client hardware. Additionally, risk to the availability of the service and data integrity appears to be a strong factor as to how much autonomy a system displays. In several cases, the highest risks seem to also be linked with the highest levels of agency [107].

Finally, tying together a fault source recommendation engine with evolutionary techniques is one of the goals of this thesis. It is the intent of the author to explore this further by making a recommendation engine using the aforementioned techniques to guide the automatic synthesis of new, valid configurations for systems once a fault is detected.

This research has demonstrated that stochastic primitives can be used to to accurately generate fault hypotheses based on feature behaviours. Also, how to prioritise and model information within stochastic primitives to exhibit specific behaviours within a system, what learning algorithms are most efficient under which circumstances, and what correlations can be discovered, if any, between multiple feature changes to correctly identify the source of a fault remain open areas for exploration. Exploration into feature locality is of particular interest as it may represent a more precise approach to reasoning the source of a fault. These topics – if explored – would build a stronger foundation upon which to establish new self-healing technologies.

APPENDIX A

APPENDIX-A

A.1 UBL Results

Results sampleed from UBL, along with the latest FDF source code and a subset of results, can be acquired at the following web-resource: *http://bit.ly/1oGBX67*.

REFERENCES

[1] J. O. Kephart, "Autonomic computing: The first decade," in *International Conference on Autonomic Computing*, (Karlsruhe, Germany), pp. 1–56, ACM SIGARCH/USENIX, 2011. New York, NY.

[2] P. Horn, "Autonomic computing: IBM's perspective on the state of information technology.," 2001.

[3] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, Issue: 1, pp. 41–50, 2003.

[4] A. G. Ganek and T. A. Corbi, "The dawning of the autonomic computing era," *IBM Systems Journal*, vol. 42 , Issue: 1, pp. 5–18, 2003.

[5] J. O. Kephart, "Research challenges of autonomic computing," (New York, NY), pp. 15–22, ACM, 2005. St. Louis, MO, USA.

[6] J. O. Kephart and W. E. Walsh, "An artificial intelligence perspective on autonomic computing policies," (Yorktown Heights, NY, USA), pp. 3–12, IEEE Computer Society, June 2004. Washington, DC, USA.

[7] C. Schneider, A. Barker, and S. Dobson, "A survey of self-healing systems frameworks," in *Software Practice and Experience*, Wiley, 2013.

[8] S. Dobson, S. Denazis, A. Fernández, D. Gaïti, E. Gelenbe, F. Massacci, P. Nixon, F. Saffre, N. Schmidt, and F. Zambonelli, "A survey of autonomic communications," *ACM Transactions on Autonomous and Adaptive Systems*, vol. 1, pp. 223–259, 2006.

[9] G. D. Rodosek, K. Geihs, H. Schmeck, and S. Burkhard, "Self-healing systems: Foundations and challenges," in *Self-Healing and Self-Adaptive Systems*, Dagstuhl Seminar Proceedings Series, (Dagstuhl, Germany), Schloß Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2009.

[10] M. Tauber, G. Kirby, and A. Dearle, "Autonomic management of maintenance scheduling in chord," *CoRR*, vol. abs/1006.1578, pp. 1–11, 2010.

[11] G. Kirby, A. Dearle, A. Macdonald, and A. Fernandes, "An approach to ad hoc cloud computing," *ArXiv.org*, 2010. http://arxiv.org/pdf/1002.4738.pdf.

[12] deeplearning.net, "Restricted boltzmann machines." http://deeplearning.net/tutorial/rbm.html, December 2014. Last Accessed: 17-June-2015.

[13] M. Carreira-Perpinan and G. Hinton, "On contrastive divergence learning," 2002. Department of Computer Science, University of Toronto.

[14] G. Hinton, S. Osindero, and Y. Teh, "A fast learning algorithm for deep belief nets," in *Neural Computation*, 2006.

[15] J. Hare, "C/.net little pitfalls: Stopwatch ticks are not timespan ticks." Blog Entry, January 2012. http://geekswithblogs.net/BlackRabbitCoder/archive/2012/01/12/c.net-little-pitfalls-stopwatch-ticks-are-not-timespan-ticks.aspx.

[16] Z. Ghahramani, "Unsupervised learning," in *Gatsby Computational Neuroscience Unit*, University College London, UK, 2004.

[17] C. Schneider, *Autonomic Techniques for Systems Management*. Sixth International Workshop on Self-Organizing Systems (IWSOS), Delft, The Netherlands, March 2012.

[18] C. Schneider, A. Barker, and S. Dobson, "Autonomous fault detection in self-healing systems: Comparing hidden markov models and artificial neural networks," in *Proceedings of International Workshop on Adaptive Self-tuning Computing Systems*, ADAPT '14, (New York, NY, USA), pp. 24:24–24:31, ACM, 2014.

[19] C. Schneider, A. Barker, and S. Dobson, "Autonomous fault detection in self-healing systems using restricted boltzmann machines," in *11th IEEE International Conference and Workshops on the Engineering of Autonomic Autonomous Systems*, (Laurel, Maryland), IEEE Computer Society, IEEE, 2014. Submitted 15 May 2014, Accepted 12 August 2014.

[20] C. Schneider, A. Barker, and S. Dobson, "Evaluating unsupervised fault detection in self-healing systems using stochastic primitives," *EAI Endorsed Transactions on Self-Adaptive Systems*, vol. 15, January 2015. doi 10.4108 sas1.1.e3.

[21] J. Kramer and J. Magee, "Self-managed systems: an architectural challenge," in *Future of Software Engineering (FOSE '07)*, (Washington, DC, USA), pp. 259 – 268, IEEE Computer Society, 2007. Minneapolis, MN.

[22] H. Psaier and S. Dustdar, "A survey on self-healing systems: approaches and systems," *Computing*, vol. 91, Issue: 1, pp. 43–73, 2010.

[23] D. Ghosh, R. Sharman, H. Raghav Rao, and S. Upadhyaya, "Self-healing systems - survey and synthesis," *Decision Support Systems*, vol. 42, pp. 2164–2185, January 2007.

[24] B. Pernici, "Self-healing systems and web services: The ws-diamond approach," in *Business Process Management Workshops*, vol. 17 of *Lecture Notes in Business Information Processing*, pp. 440–442, Springer Berlin Heidelberg, 2009.

[25] A. Viterbi, "Error bounds for convolutional codes and an asymptotically optimum decoding algorithm," *IEEE Transactions on Information Theory*, vol. 13, pp. 260–269, April 1967.

[26] A. J. Ramirez, D. B. Knoester, B. H. Cheng, and P. K. McKinley, "Applying genetic algorithms to decision making in autonomic computing systems," in *Proceedings of the 6th international conference on Autonomic computing*, ICAC '09, (New York, NY, USA), pp. 97–106, ACM, 2009.

[27] A. J. Ramirez, D. B. Knoester, B. H. Cheng, and P. K. Mckinley, "Plato: a genetic algorithm approach to run-time reconfiguration in autonomic computing systems," *Cluster Computing*, vol. 14, pp. 229–244, September.

[28] D. J. Dean, H. Nguyen, and X. Gu, "Ubl: Unsupervised behavior learning for predicting performance anomalies in virtualized cloud systems," in *Proceedings of the 9th international conference on Autonomic computing*, ICAC '12, (New York, NY, USA), pp. 181–190, ACM, 2012.

[29] L. Prodan, G. Tempesti, D. Mange, and A. Stauffer, "Embryonics: artificial stem cells," in *In: Proc. of ALife VIII*, pp. 101–105, Cambridge, MA, USA: MIT Press, 2002.

[30] B. J. Garvin, M. B. Cohen, and M. B. Dwyer, "Using feature locality: Can we leverage history to avoid failures during reconfiguration?," in *Proceedings of the 8th Workshop on Assurances for Self-adaptive Systems*, ASAS '11, (New York, NY, USA), pp. 24–33, ACM, 2011. Szeged, Hungary.

[31] B. Garvin, M. Cohen, and M. Dwyer, "Failure avoidance in configurable systems through feature locality," vol. 7740, pp. 266–296, 2013.

[32] C. Schuler, R. Weber, H. Schuldt, and H. j. Schek, "Scalable peer-to-peer process management - the osiris approach," in *In: Proceedings of the 2nd International Conference on Web Services (ICWS '2004)*, (San Diego, CA), pp. 26–34, IEEE Computer Society, 2004. Washington DC, USA.

[33] I. Stoica, R. Morris, D. Karger, and M. F. Kaashoek, "Chord: A scalable peer-to-peer lookup service for internet," in *Proceedings of the ACM SIGCOMM '01 Conference*, (San Diego, CA), pp. 1–12, ACM, 2001. New York, NY.

[34] N. Stojnic and H. Schuldt, "Osiris-sr: A safety ring for self-healing distributed composite service execution," in *Software Engineering for Adaptive and Self-Managing Systems (SEAMS), 2012 ICSE Workshop on*, (Zürich, Switzerland), pp. 21–26, ACM, 2012. New York, NY.

[35] H. Naccache, G. Gannod, and K. Gary, "A self-healing web server using differentiated services," in *Service-Oriented Computing - ICSOC 2006*, vol. 4294 of *Lecture Notes in Computer Science*, pp. 203–214, Springer Berlin / Heidelberg, 2006.

[36] D. Miorandi, I. Carreras, E. Altman, L. Yamamoto, and I. Chlamtac, "Bio-inspired approaches for autonomic pervasive computing systems," in *Bio-Inspired Computing and Communication*, vol. 5151, pp. 217–228, Springer Berlin, 2008.

[37] D. Miorandi, D. Lowe, and L. Yamamoto, "Embryonic models for self-healing distributed services," in *Bioinspired Models of Network, Information, and Computing Systems*, vol. 39 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pp. 152–166, Springer Berlin Heidelberg, 2010.

[38] D. Menasce, H. Gomaa, S. Malek, and J. Sousa, "Sassy: A framework for self-architecting service-oriented systems," *Software, IEEE*, vol. 28, no. 6, pp. 78–85, 2011.

[39] R. Calinescu, "General-purpose autonomic computing," in *Autonomic Computing and Networking*, pp. 3–30, Springer US, 2009.

[40] M. Salehie and L. Tahvildari, "Self-adaptive software: Landscape and research challenges," *ACM Trans. Auton. Adapt. Syst.*, vol. 4, pp. 14:1–14:42, May 2009.

[41] D. D. Clark, C. Partidge, J. C. Ramming, and J. T. Wroclawski, "A knowledge plane for the iternet," in *Proceeding of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM 03, (New York, NY, USA), pp. 3–10, ACM, 2003. doi: 10.1145 - 863955.863957.

[42] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach. 2nd Edition*. Prentice Hall, 2003.

[43] Y. Diao, J. L. Hellerstein, S. Parekh, R. Griffith, G. Kaiser, and D. Phung, "Self-managing systems: A control theory foundation," *Engineering of Computer-Based Systems*, vol. 23, pp. 2213–2222, 2005.

[44] M. Brodie, S. Ma, G. Lohman, T. Syeda, L. Mahmood, N. Mignet, Modani, M. Wilding, J. Champlin, and P. Sohn, "Quickly finding known software problems via automated symptom matching.," (Washington, DC, USA), pp. 101–110, IEEE Computer Society, 2005. Seattle, WA.

[45] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox, "Microreboot-a technique for cheap recovery," vol. 6, (Berkeley, CA, USA), p. 3, USENIX Association, 2004. San Francisco, CA.

[46] D. E. Irwin, L. E. Grit, and J. Chase, "Balancing risk and reward in market-based task scheduling," (Honolulu, HI), pp. 160–169, IEEE Proceedings 2004, 2004. Washington, DC, USA.

[47] C. Boutilier, R. Das, J. O. Kephart, G. Tesauro, and W. E. Walsh, "Cooperative negotiation in autonomic systems using incremental utility elicitation," (San Francisco, CA, USA), pp. 89–97, Morgan Kaufmann Publishers Inc., 2003. Aalborg, Denmark.

[48] R. Braynard, D. Kostic, A. Rodriguez, J. Chase, and A. Vahdat, "Opus: an overlay peer utility service.," in *In Proceedings of the 5th International Conference on Open Architectures and Network Programming (OPENARCH)*, (New York, NY), pp. 167 – 178, IEEE Communications, 2002. Atlanta, GA. USA.

[49] W. E. Walsh, G. Tesauro, J. O. Kephart, and R. Das, "Utility functions in autonomic systems," in *Proceedings of the First International Conference on Autonomic Computing*, (Washington, DC, USA), pp. 70–77, IEEE Computer Society, 2004. New York, NY, USA.

[50] J. Pearl, *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan Kaufmann, 1988.

[51] G. Casella and E. George, "Explaining the gibbs sampler," *The American Statistician*, vol. 46, no. 3, pp. 167–174, 1992.

[52] A. Fischer and C. Igel, “An introduction to restricted boltzmann machines,” in *Progress in Pattern Recognition, Image Analysis, Computer Vision, and Applications*, vol. 7441 of *Lecture Notes in Computer Science*, pp. 14–36, Springer Berlin Heidelberg, 2012.

[53] O. Tibermacine, C. Tibermacine, and F. Cherif, “A process to identify relevant substitutes for healing failed ws-* orchestrations,” *Journal of Systems and Software*, 2015. Elsevier.

[54] E. U. Warriach, T. Ozcelebi, and J. J. Lukkien, “Self-* properties in smart environments: Requirements and performance metrics,” in *Workshop Proceedings of the 10th International Conference on Intelligent Environments*, p. 194, IOS Press, 2014.

[55] Q. Shen, J. Cao, and H. Gu, “A similarity network based behavior anomaly detection model for computer systems,” in *Computational Science and Engineering (CSE), 2014 IEEE 17th International Conference on*, pp. 1738–1745, IEEE, 2014.

[56] M. Minksey and S. Papert, *An Introductions to Computational Geometry*. MIT Press, 1969. ISBN 0-262-63022-2.

[57] R. R. Schaller, “Moore’s law: Past, present, and future,” *IEEE Spectr.*, vol. 34, pp. 52–59, June 1997. DOI 10.1109 6.591665.

[58] J. Biolchini, P. G. Mian, A. C. C. Natali, and G. H. Travassos, “System engineering and computer science department coppe/ufrj, technical report es,” Tech. Rep. 05, 2005.

[59] B. Kitchenham, “Procedures for undertaking systematic reviews,” tech. rep., Keele University and National ICT Australia Ltd., 2004.

[60] B. Kitchenham, O. Pearl Brereton, D. Budgen, M. Turner, J. Bailey, and S. Linkman, “Systematic literature reviews in software engineering - a systematic literature review,” *Inf. Softw. Technol.*, pp. 7–15, 2009. Newton, MA, USA.

[61] B. Kitchenham, T. Dybaß, and M. Jørgensen, “Evidence-based software engineering,” in *roceedings of the 26th International Conference on Software Engineering (ICSEŠ04)*, (Washington DC, USA), IEEE Computer Society, 2004.

[62] Y. Bengio, E. Thibodeau-Laufer, and J. Yosinski, “Deep generative stochastic networks trainable by backprop,” in *Proceedings of the Thirty-one International Conference on Machine Learning (ICML’14)*, Springer, 2014.

[63] L. Rilling, “Vigne: Towards a self-healing grid operating system,” in *Euro-Par 2006 Parallel Processing*, vol. 4128 of *Lecture Notes in Computer Science*, pp. 437–447, Springer Berlin / Heidelberg, 2006.

[64] M. Sloman, "Policy driven management for distributed systems," *Journal of Network and Systems Management*, vol. 2, pp. 333–360, 1994.

[65] S.-W. Cheng, A.-C. Huang, D. Garlan, B. R. Schmerl, and P. Steenkiste, "Rainbow: Architecture-based self-adaptation with reusable," in *ICAC*, (New York, NY, USA), pp. 276–277, IEEE Computer Society, 2004. New York, NY.

[66] S.-W. Cheng, D. Garlan, and B. Schmerl, "Architecture-based self-adaptation in the presence of multiple objectives," in *ICSE 2006 Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, (Shanghai, China), pp. 2–8, ACM, 2006. New York, NY.

[67] J. Simmonds, S. Ben-David, and M. Chechik, "Monitoring and recovery of web service applications," in *The Smart Internet*, vol. 6400 of *Lecture Notes in Computer Science*, pp. 250–288, Berlin, Germany: Springer-Verlag, 2010.

[68] S. Ahmed, S. I. Ahamed, M. Sharmin, and C. S. Hasan, "Self-healing for autonomic pervasive computing," in *Autonomic Communication*, pp. 285–307, Springer US, 2009.

[69] M. Aldinucci, M. Danelutto, G. Zoppi, and P. Kilpatrick, "Advances in autonomic components and services," in *From Grids to Service and Pervasive Computing* (T. Priol and M. Vanneschi, eds.), pp. 3–17, Springer US, 2008.

[70] V. Cardellini, E. Casalicchio, V. Grassi, S. Iannucci, F. Lo Presti, and R. Mirandola, "Moses: A framework for qos driven runtime adaptation of service-oriented systems," *IEEE Transactions on Software Engineering*, vol. PP, no. 99, pp. 1–23, 2011.

[71] H. Psaier, F. Skopik, D. Schall, and S. Dustdar, "Behavior monitoring in self-healing service-oriented systems," in *Socially Enhanced Services Computing*, pp. 95–116, Springer Vienna, 2011.

[72] O. Shehory, *A Self-healing Approach to Designing and Deploying Complex, Distributed and Concurrent Software Systems*, vol. 4411 of *Lecture Notes in Computer Science*, pp. 3–13. Springer-Verlag, 2007.

[73] G. Li, L. Liao, D. Song, J. Wang, F. Sun, and G. Liang, "A self-healing framework for qos-aware web service composition via case-based reasoning," in *Web Technologies and Applications*, vol. 7808 of *Lecture Notes in Computer Science*, pp. 654–661, Springer Berlin Heidelberg, 2013.

[74] A. Carzaniga, A. Gorla, and M. Pezzè, "Healing web applications through automatic workarounds," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 10, pp. 493–502, 2008. 10.1007/s10009-008-0088-8.

[75] S. Hassan, D. McSherry, and D. Bustard, "Autonomic self healing and recovery informed by environment knowledge," *Artificial Intelligence Review*, vol. 26, pp. 89–101, 2006. 10.1007/s10462-007-9033-6.

[76] S. Kamvar, M. Schlosser, and H. Garcia-Molina, "The eigentrust algorithm for reputation management in p2p networks," in *Proceedings of the 12th international conference on World Wide Web*, WWW '03, (New York, NY, USA), pp. 640–651, ACM, 2003.

[77] D. M. Chess, "Security in autonomic computing," vol. 33, 2005.

[78] R. Gustavsson and B. Ståhl, "Self-healing and resilient critical infrastructures," in *Critical Information Infrastructure Security*, vol. 5508 of *Lecture Notes in Computer Science*, pp. 84–94, Springer Berlin / Heidelberg, 2009.

[79] L. Baduel and S. Matsuoka, "A decentralized, scalable, and autonomous grid monitoring system," in *Principles of Distributed Systems*, vol. 4878 of *Lecture Notes in Computer Science*, pp. 1–15, Springer Berlin / Heidelberg, 2007.

[80] J. H. Holland, "Adaptation in natural and artificial systems," *MIT Press, Cambridge, MA. US.*, 1992.

[81] A. Metzger, O. Sammodi, and K. Pohl, "Accurate proactive adaptation of service-oriented systems," in *Assurances for Self-Adaptive Systems* (J. Cãmara, R. a. Lemos, C. Ghezzi, and A. a. Lopes, eds.), vol. 7740 of *Lecture Notes in Computer Science*, pp. 240–265, Springer Berlin Heidelberg, 2013.

[82] J. Fernandez-Marquez, G. Di Marzo Serugendo, and S. Montagna, "Bio-core: Bio-inspired self-organising mechanisms core," in *Bio-Inspired Models of Networks, Information, and Computing Systems*, vol. 103 of *Lecture Notes of the Institute for Computer Sciences*, pp. 59–72, Berlin, Germany: Springer Berlin Heidelberg, social informatics and telecommunications engineering ed., 2012. Social Informatics and Telecommunications Engineering Volume.

[83] S. Montagna, D. Pianini, and M. Virolio, "Gradient-based self-organisation," in *6th IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO 2012)*, (Washington DC, USA), pp. 10–14, IEEE Computer Society, 2012. Lyon, France.

[84] J. McCann, R. de Lemos, M. Heubscher, F. O. Rana, and A. Wombacher, "Can self-managed systems be trusted? some views and trends," *Knowledge Engineering Review*, vol. 21, pp. 239–248, September 2006.

[85] J. McCann and M. Huebscher, "Evaluation issues in autonomic computing," in *Grid and Cooperatve Computing - GCC 2004 Workshops*, vol. 3252, pp. 597–608, Springer Berlin, 2004.

[86] R. de Lemos, "The conflict between self-* capabilities and predictability," in *Self-star Properties in Complex Information Systems*, vol. 3460 of *Lecture Notes in Computer Science*, pp. 218–228, Spinger Berlin Heidelberg, 2005.

[87] H. Gomaa and K. Hashimoto, "Dynamic self-adaptation for distributed service-oriented transactions," in *Software Engineering for Adaptive and Self-Managing Systems (SEAMS), 2012 ICSE Workshop on*, (Zürich, Switzerland), pp. 11–20, ACM, 2012. Washington, DC, USA.

[88] Y. Engel and O. Etzion, "Towards proactive event-driven computing," in *Proceedings of the 5th ACM international conference on Distributed event-based system*, DEBS '11, (New York, NY, USA), pp. 125–136, ACM, 2011.

[89] C. Ortega-Sanchez, M. Mange, S. Smith, and A. Tyrrell, "Embryonics: a bio-inspired cellular architecture with fault-tolerant properties," in *Genetic Programming and Evolvable Machines*, vol. 1, pp. 187–215, Dordrecht, the Netherlands: Kluwer Academic Publishers, 2000.

[90] T. Kohonen, "The self-organizing map," *Proceedings of the IEEE*, vol. 78, no. 9, pp. 1464–1480, 1990.

[91] Z. Zheng, L. Yu, Z. Lan, and T. Jones, "3-dimensional root cause diagnosis via co-analysis," in *Proceedings of the 9th international conference on Autonomic computing*, ICAC '12, (New York, NY, USA), pp. 181–190, ACM, 2012.

[92] Y. Dai, Y. Xiang, and G. Zhang, "Self-healing and hybrid diagnosis in cloud computing," in *Cloud Computing*, vol. 5931 of *Lecture Notes in Computer Science*, pp. 45–56, Springer Berlin / Heidelberg, 2009.

[93] P. Snyder, G. Valetto, J. Fernandez-Marquez, and G. di Marzo Serugendo, "Augmenting the repertoire of design patterns for self-organized software by reverse engineering a bio-inspired p2p system," in *Proceedings of the 6th IEEE International Conference on*

Self-Adaptive and Self-Organizing Systems (SASO 2012), (Lyon, France), pp. 199–204, IEEE Computer Society, September 2012. Washington, DC, USA.

[94] D. M. Chess, V. Kumar, A. Segal, and I. Whalley, "Work in progress: Availability-aware self-configuration in autonomic systems," in *Utility Computing*, vol. 3278 of *Lecture Notes in Computer Science*, pp. 257–258, Springer Berlin / Heidelberg, 2004.

[95] I. E. Fellows, "Why (and when and how) contrastive divergence." ArXiv.org, May 2014. http://arxiv.org/pdf/1405.0602v1.pdf.

[96] L. Baum and T. Petrie, "Statistical inference for probabilistic functions of finite state markov chains," *The Annals of Mathematical Statistics*, vol. 37, no. 6, pp. 1554–63, 1966.

[97] L. Baum and T. Petrie, "An inequality with applications to statistical estimation for probabilistic functions of markov processes and to a model for ecology," *Bulletin of the American Mathematical Society*, vol. 73, no. 3, pp. 360–3, 1967.

[98] L. Baum and T. Petrie, "A maximization technique occurring in the statistical analysis of probabilistic functions of markov chains," *The Annals of Mathematical Statistics*, vol. 41, no. 1, pp. 164–71, 1970.

[99] H. Schulz, A. Müller, and S. Behnke, "Investigating convergence of restricted boltzmann machine learning," in *NIPS 2010 Workshop on Deep Learning and Unsupervised Feature Learning*, 2010.

[100] E. J. Humphrey, J. P. Bello, and Y. LeCun, "Moving beyond feature design: Deep architectures and automatic feature learning in music informatics," in *ISMIR* (F. Gouyon, P. Herrera, L. G. Martins, and M. Müller, eds.), pp. 403–408, FEUP Edições, 2012. ISBN: 978-972-752-144-9.

[101] A. Kirillov, "Aforge.net framework." http://www.aforgenet.com/framework/members.html, 2013.

[102] C. R. Souza, "Accord.net framework," 2013. http://accord-framework.net/.

[103] K. Soska and N. Christin, "Automatically detecting vulnerable websites," in *23rd USENIX Security Symposium.*, (San Diego, CA), USENIX, 2014.

[104] Y. Tan, H. Nguyen, Z. Shen, X. Gu, C. Venkatramani, and D. Rajan, "Prepare: Predictive performance anomaly prevention for virtualized cloud systems," in *ICDCS'12*, pp. 285–294, 2012.

[105] A. Keller and M. Brunner, "Self-managing systems and networks," *Journal of Network and Systems Management*, vol. 13, pp. 147–149, 2005. 10.1007/s10922-005-4438-5.

[106] A. Burnap, Y. Ren, H. Lee, R. Gonzalez, and P. Papalambros, "Improving preference prediction accuracy with feature learning," in *Proceedings of the ASME 2014 International Design Engineering Technical Conferences Computers and Information in Engineering Conference* (N. Y. U. S. August 17-20, 2014, ed.), DETC/CIE 2014, ASME, 2014.

[107] G. Brady, R. Sterrit, and G. Wilkie, "An adaptive approach to self-healing in an intelligent environment," in *Proceedings for ADAPTIVE 2014, The Sixth International Conference on Adaptive and Self-Adaptive Systems and Applications*, IARIA, May 2014. ISBN 978 1 61208 341 4.