Error Diagnosis of Cloud Application Operation Using Bayesian Networks and Online Optimisation

Xiwei Xu, Liming Zhu, Daniel Sun, An Binh Tran, Ingo Weber, Min Fu, Len Bass
SSRG, NICTA, Sydney, Australia
School of Computer Science and Engineering, UNSW, Sydney, Australia
{firstname.lastname}@nicta.com.au
REGULAR PAPER

***Abstract*—Operations such as upgrade or redeployment are an important cause of system outages. Diagnosing such errors at runtime poses significant challenges. In this paper, we propose an error diagnosis approach using Bayesian Networks. Each node in the network captures the potential (root) causes of operational errors and its probability under different operational contexts. Once an operational error is detected, our diagnosis algorithm chooses a starting node, traverses the Bayesian Network and performs assertion checking associated with each node to confirm the error, retrieve further information and update the belief network. The next node in the network to check is selected through an online optimisation that minimises the overall availability risk considering diagnosis time and fault consequence. Our experiments show that the technique minimises the risk of faults significantly compared to other approaches in most cases. The diagnosis accuracy is high but also depends on the transient nature of a fault.**

Keywords*—*Diagnosis; Cloud; Operation; Bayesian network; Online Optimisation

I. Introduction

Cloud applications are subject to many types of sporadic system operations, such as upgrade, reconfiguration, on-demand scaling, deployment, and auto-recovery from node failures. Both the ease of such operations (programmable through Cloud infrastructure APIs) and the prevalence of continuous deployment practices are significantly increasing the frequency and simultaneousness of these operations. Recent empirical research [1] has shown that the majority of system-wide failures (i.e., outages) are caused by system operations, such as starting a service, adding a node to a running system, or restarting a node. Among the 9 categories of causes (input events) of system-wide failures, 69% are caused by operation-related activities. According to the same study, a large portion of outages are also caused by automated, cascading overreactions to initial small errors – usually because the cause was not known and the automated fault-tolerance reactions simply replaced erroneous resources, rather than fixing the cause. These observations were confirmed by another large-scale empirical study [2] of 3000 issues in cloud systems. Most issues come from under-tested "operation protocols" (cloning, replication and recovery) rather than the primary features of the application. Also, a survey of 50 system administrators from multiple countries concluded that 8.6% of operations fail [3]. Some of these failures are due to unreliable Cloud infrastructure APIs [4], [5]. Our own *interviews with industry* on outages (unpublished as yet) revealed several specific reasons. For example, one outage happened because newly replaced VMs in an auto-scaling group (ASG) did not have an application installed properly. The corresponding health check was misconfigured, which led to more and more healthy VMs being replaced by broken ones. Another outage happened because VMs were terminated based on their age rather than current workload, causing many heavily loaded VMs to be terminated.

Diagnosing these errors quickly, especially at runtime to allow better cause-aware recovery actions, poses significant challenges. Most error diagnosis heavily relies on an operator's experiences, her familiarity with the system and a loosely ordered diagnostic path. An operator would manually issue a sequence of commands to acquire more information and assert whether a particular candidate cause is the actual cause, until an actionable (root) cause is identified. She would also dig through logs and observe metrics and dashboards. This takes a significant amount of time. Meanwhile the failure persists, as does the cause, and cause-unaware recovery potentially keeps doing more harm than good. This might be inevitable if the errors and the causes are specific to the application itself. However, we have observed that:

1) Operations of application usually have a relatively smaller set of well-understood faults, especially if the operation is manipulating course-grained VMs, lightweight containers, and application binaries, and if the configuration of them uses standard infrastructure facilities and APIs.
2) The operational context is an important input to diagnosis: it can guide diagnosis in the right direction. During specific parts of operations, certain types of (unreliable) APIs or activities are exercised more frequently. Therefore, an otherwise unlikely fault can suddenly become the most likely cause.
3) Error diagnosis takes time and can follow different diagnostic paths. Deciding on the most efficient diagnostic path to take should consider both the likelihood of the causes and the consequences, so as to minimise the risk.

Motivated by these observations, in this paper we propose a novel approach for automated diagnosis of non-transient faults during operations, i.e., faults that are still present when the diagnosis checks if they are present. The approach consists of two parts: (i) a model of common operational issues, i.e., faults/errors/failures, along with contextual, probabilistic, and risk information about them; and (ii) an optimisation algorithm to select the best diagnostic path when an issue occurs. In more detail, we use a Bayesian Network (BN) to model operational

issues as nodes in BN, the probability of them happening and the dependencies among them. Each node also has a set of assertions that can be run to confirm the presence of this issue at runtime, or to retrieve further information required for future diagnostics. The online optimisation algorithm we devised re-evaluates the situation after each step, and selects the next nodes whose assertions to check in a manner that minimises risk, e.g., that a significant fault is left undiagnosed for too long. In the past, Fault Trees (FTs) [6] were used to assist diagnosis without considering fault probability or risk. Our approach starts by converting such a FT to a BN, since the BN enables belief updates using runtime information and the consideration of the above-mentioned additional factors.

The main contributions of this paper are:

- A BN-based model of operational issues (faults/errors /failures) and its association with assertion checking to enable runtime confirmation of the presence of issues.
- Construction of a BN from a FT, with prior probabilities for root causes collected from literature, historical data, and BN-driven derivation.
- A diagnosis algorithm that traverses a BN to execute assertion checking associated with BN nodes, and update belief from the assertion results. The algorithm also incorporates the concept of soft evidence to accommodate potential false positives/negatives of assertion results.
- An online optimisation algorithm that chooses a diagnosis path that minimises risk within acceptable diagnosis time.

Our evaluation uses real outages encountered by industry, as captured in our interviews mentioned above. Comparative experiments show that the technique minimises the risk significantly compared to other approaches and meanwhile, maintains comparable diagnosis time. For non-transient, known faults in our evaluation, we can achieve perfect diagnosis accuracy.

The paper outlines background information in Section II, followed by an overview of our approach in Section III. Section IV presents the construction of BNs from FTs, and Section V describes the online diagnosis, including belief updates, the diagnosis algorithm, and the diagnosis path selection through optimisation. We evaluate our approach in Section VI and discuss the limitations in Section VII. Related work is discussed in Section VIII, and Section IX concludes the paper.

II. Background

A. Bayesian Networks

BNs [7] are graphical models for reasoning about a domain under uncertainty. The structure of the network captures qualitative relationships between variables. In BNs, nodes represent variables (discrete or continuous) and directed arcs represent causal connection between variables.

BNs are used to predict the probability of unknown variables and to update the probabilities when new knowledge arrives. In particular, when new values of some nodes are observed, the probability of all the nodes can be updated by applying Bayes' Theorem. This process is called *probability propagation* (or *inference* or *belief update*). Such new information is called *evidence*. There are different types of evidence: specific evidence, negative evidence, and virtual/soft evidence. Soft evidence is used to reflect the uncertainty in the evidence itself. For example, if you should update a BN based on the evidence that someone tells you she does not smoke, there is a small chance she is lying. We use soft evidence to reflect that runtime diagnostic assertions may return false results.

BNs and FTs are the most widely used techniques for dependability analysis. In comparison, BNs allow for more powerful modelling and analysis. FTs are limited through their static structure and in their uncertainty handling [8].

B. System Operation and Process Context

Applications in Cloud are subject to sporadic changes due to operational activities such as upgrade, snapshotting, redeployment, restart, recovery, and replication among others. Some operations can be relatively long-lasting such as large-scale upgrade and reconfiguration. Multiple operations can happen at the same time. There is a sporadic nature to these operations, as some are triggered by ad hoc bug fixing and feature delivery while others are triggered periodically. Some operations have significant impact at both the node level and the system level. For example, upgrading could affect overall system capacity depending on the upgrade strategy, since an individual virtual machine (VM) may be rebooted or replaced. The types of possible faults and their likelihood during such operations can be dramatically different when compared to normal operation.

Our previous work took a process view of an operation [6]. Process context consists of the operations process id, instance id, the start and end time of operations, and their characteristics. Such process context can be captured in our BN model. The process models can be automatically mined from the logs produced by the operation [9]. Along with the process model, a set of regular expressions associated with each of the step was generated to enable runtime mapping from observed log lines to the current step in the process. Such detection of operation progression enables us to take process context into consideration. At runtime, the process context information can be used in the BN model as evidence to adjust the occurrence probability associated with nodes – e.g., during a step when VMs are rebooted, errors related to that are much more likely.

III. Overview of the Diagnosis Approach

In this work, we want to consider fault occurrence probabilities, fault consequences, and the impact of diagnosis time combined with fault consequence. We explain how BNs are used to capture the probabilistic causal relationships between context, faults, and errors, how updates are handled, and how an online optimisation algorithm can minimise risk when choosing a diagnosis path.

Fig. 1 gives a graphical overview of our new approach. Firstly a BN is generated *offline*, either manually or from an existing FT. In our case, the input FT is constructed based on our knowledge of common cloud operational faults, Cloud API faults, and system outages from real-world industry scenarios. Compared with BN, modelling an FT is more intuitive, and organisations may have existing FTs as well as modelling expertise. Thus, we devised a mapping algorithm – discussed in Section IV – that converts a FT into a BN and enriches it with information such as (process) context and probabilities.

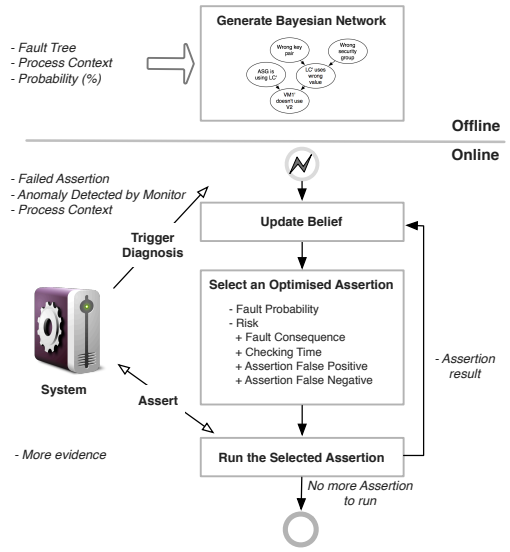

Fig. 1: Overview of Error Diagnosis Workflow.

The *online* diagnosis process is triggered when an anomaly is detected by some monitoring mechanism. The detected anomaly and the current process context are the first evidence, based on which the BN updates the beliefs over the whole network, to adjust the occurrence probabilities of the intermediate errors and root causes. Details of belief updates are discussed in Section V-A.

After the first belief updates, the set of BN nodes corresponding to the most urgent causes is determined, and assertion checking for these nodes is started. Then, as the more and more assertions return results, the belief is further updated based on this new evidence. The diagnosis follows an algorithm traversing the network. At any given time, the algorithm maintains a *to-check list* of BN nodes: when the value of any of the nodes becomes known (by way of assertion checking) that confirms or denies the existence of particular faults or fault groups. The current to-check list can be a sub-graph of the BN. It, too, is updated every time new evidence is received. The diagnosis terminates when there is no node left in the to-check list, or an actionable (root) cause is identified. It is possible that no fault is found after the diagnosis stops. In this case, the error that was initially detected is caused by either a transient fault or a false positive in error detection, or the diagnosis failed due to an incomplete BN. The details of the diagnosis algorithm are discussed in Section V-B.

There are different *types of assertions*. Assertions can be on-demand diagnostic checks that query the current state of resources or systems (e.g., issuing a health check command on the spot or querying the public cloud provider API for the current states of cloud resources). They can acquire information from logs or other systems that already captured the desired information. Assertions may also involve human operators, where the operator performs a check and provides the result of the assertion subsequently. On any path through the BN, a mix of these types of assertions may be encountered. Following the path to confirm a particular fault may take considerable time, especially if a human needs to be in the loop.

It can be the case that multiple *alternative paths* could diagnose a fault. Choosing a diagnosis path is done based on the outcome of the optimisation algorithm below. Some assertions can proceed simultaneously, while others have to be checked in sequence – e.g., when one assertion relies on information obtained by another assertion. It is usually not feasible to assert the root causes directly due to assertion dependency, the very large state space of the whole system, and the performance impact of investigating a larger state space. Just like a human operator, the semi-automated diagnosis usually follows few paths with limited concurrency. Where an opportunity for concurrent checking exists, we could make use of it.

We use an *online optimisation algorithm* to choose the diagnosis path that presents the best trade-off, given the current information. The optimisation algorithm selects the next assertion(s) for the to-check list so that the diagnosis has the minimal risk through excluding the faults with higher risk first. The risk is calculated based on several factors including the probability and consequence of faults, the time required for assertion checking to confirm a fault, and false positive/negative rates of the respective assertion checking. Details of the optimisation algorithm are discussed in Section V-C.

IV. Bayesian Network Construction

Our approach starts from a FT, and we produce a BN from it. This is not the only pair of notations one could use, it is one useful choice, and we demonstrate its effectiveness. The initial notation should be one that is widely used by practitioners (so existing models in organizations can be used), and it should facilitate communication among a range of stakeholders; Fault Trees have those features.

Mapping methods exist in the literature [8], [10], as do tools that support the conversion [11], [12]. We selected an existing mapping method [8] and adapted it to integrate the modelling of (process) context. Fig. 2 illustrates the procedure used to construct the graphical structure of a BN from a FT and to generate prior and conditional probabilities for each variable. Currently, the one-off effort of converting a FT to a BN is done manually. We use Netica-J[1] to construct the BN.

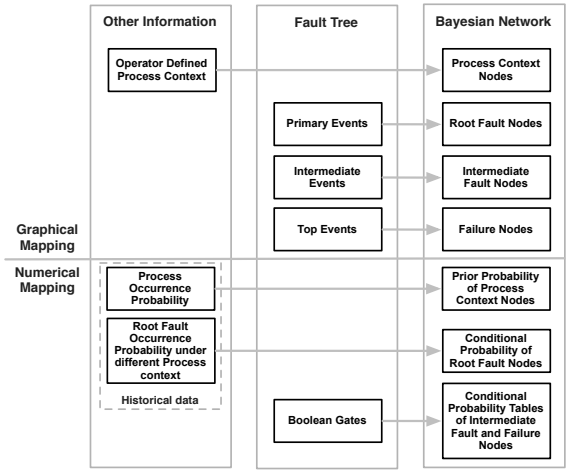

Fig. 2: Construction of Bayesian Network.

[1] http://www.norsys.com

A. Running Example

To illustrate our approach, we introduce a representative error/fault scenario. Say, one VM (or VM instance, or just instance) is not using the expected version of an application. Such an error might cause anomalies at the application level or access problems. The example assumes the problematic VM is within an auto-scaling group (ASG), which is a common deployment configuration on AWS (Amazon Web Services)[2]. ASGs help maintaining the availability of an application by automatically scaling the capacity in or out, according to user-defined conditions. An ASG is always associated with a Launch Configuration (LC), which specifies how to launch VMs for the ASG. The LC includes information such as the Amazon Machine Image (AMI, i.e., from which VM image to launch new VMs), ssh key pairs, security groups (SGs), and other configuration settings. An SG defines firewall rules which control the traffic to or from the associated VMs.

Fig. 3 shows the relevant FT sub-tree for this scenario. The wrong application version in the problematic VM could be caused by an ill-configured LC (left-most branch). If the current LC (denoted as LC') is not the known correct LC, that may or may not be the root cause. To confirm LC' is correct, all its fields need to have the same values as in LC. Other problems relate to settings for key pairs, SGs, and AMIs.

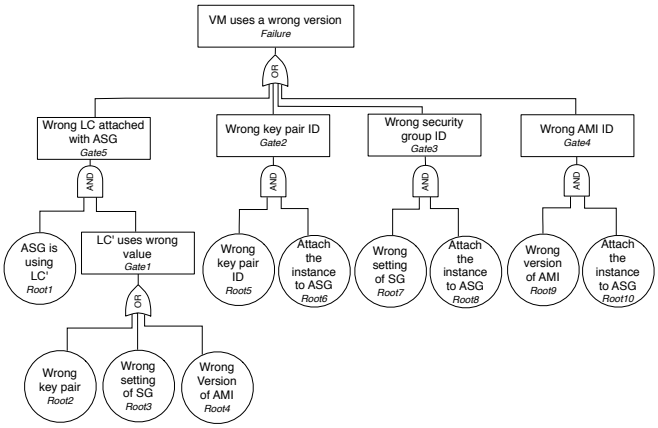

Fig. 3: Fault Tree of the running example.

B. Graphical Mapping

In graphical mapping, the primary, intermediate, and top events of the FT are converted into the root fault nodes, intermediate fault nodes, and failure nodes in BN. Taking the semantics of the primary events and Boolean logics from the FT, all resulting BN nodes are binary: True or False. Fig. 4 shows the BN from converting the sub-tree given above. The nodes in the BN are connected in the same way as the corresponding events in the FT.

We model process context (PC) as a set of nodes – shown in blue in Fig. 4. Each process context node represents whether a process or a major step of a process is currently executed or not. The figure shows two examples: the solid blue node models rolling upgrade, and the faded blue node models another process context that affects VM-level faults. The system operator decides what types of processes are

[2]http://aws.amazon.com

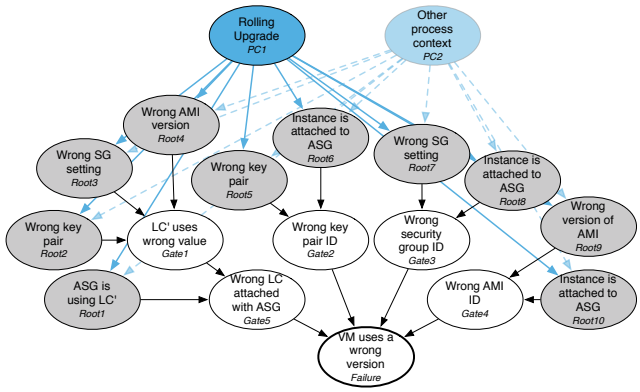

Fig. 4: Bayesian Network of the running example.

interesting enough to be explicitly modelled in the BN, and what the proper granularity, typically based on the impact of a certain process or step on the occurrence probability of the faults. Other contexts of interest can be modelled using the same method.

The arrows between the process context nodes and the root fault nodes represent the causal relationship between the process/step and the occurrence of the root faults. The effect of the process context on the occurrence of the intermediate faults and failure is implied by the conditional probability. Context nodes are binary with two values: on or off.

C. Probability Acquisition

The prior probabilities for some of the root faults and (process) context can be taken from the documentation of commercial products or from the literature, e.g., experiments [5] and empirical studies [1], [3], [4], [13]. Prior probabilities of remaining root faults can be calculated based on system operation scenarios. We classified the (root) causes into three main categories: human error, virtual resource failure, and API failure. We do not consider hardware failures separately in our approach, since these are masked to the consumers of public clouds and thus covered by the probability of virtual resource failure. Below we show how to calculate prior probabilities for average operators and assuming AWS cloud services. For different environments, different base data needs to be used.

For **Human error**, we refer to the examples of human error probabilities included in [14]. In the context of system operation, human interactions with the system occur during configuration, operation, diagnosis and system recovery. We only consider configuration and operation since our scope is error diagnosis after a system fails. We assume 8.6% of system operations fail based on a survey from multiple countries [3], where most of the participants had more than 5 years of experience. An extensive set of experiments are conducted in [13] to better understand the occurrence probability, category, and impact of human errors. They identified 7 fault categories from 42 faults found in 43 conducted experiments. 5 out of 7 fault categories are within our scope: unnecessary restart of SW component, start of wrong SW version, incorrect restart, global misconfiguration, local misconfiguration. The probability of a certain type of fault is estimated by multiplying the occurrence rate of the fault and 8.6%.

Virtual resource failure is a common problem on cloud infrastructure due to various sources of uncertainty. We assume the availability of AWS EC2 is the same for AWS availability zones at 99.95%, and the availability of ELB(Elastic Load Balancer) and AMI(Amazon Machine Images) to be the same as S3(Simple Storage Service) at 99.99%. The failure probability is $1 - availability$. The age of a resource also has impact on its failure probability. In an empirical study investigating the faults causing catastrophic failures on distributed systems [1], one finding shows that a recently restarted/added node has the higher failure probability compared to other running nodes. Among the identified 9 categories of input events (causes) to failures, starting/restarting/adding a new node together cover 45.3% of all failures [1][Table 4]. Thus, we distinguish recently manipulated nodes from other running nodes with different prior probabilities, by increasing failure probability for them.

Cloud API failures represent a large percentage of errors/failures on cloud applications. For example, more than half (53%) of the error cases reported in Amazon's EC2 forum are related to API failures [4]. We assume that the probability of API failure is *P(API failure) = 53%*8.6%=4.6%*.

The **relationships** between connected nodes are quantified through assigning conditional probability distributions. In our scenario we only consider discrete variables, and thus distributions take the form of conditional probability tables (CPTs). Table I gives a sample CPT for root fault node *ASG is using LC'*. The parent set of the root fault nodes contains the context nodes PC1 and PC2. In the CPT, the probability for both values (True or False) for the chosen node is symbolised as P_i, under the condition of one of the possible combinations of values of PC1 and PC2. The LC associated with ASG is more likely to be changed during rolling upgrade, thus $P_1 > P_3$ and $P_2 > P_4$.

TABLE I: Sample CPT of root fault nodes

Rolling Upgrade (PC1)	On	On	Off	Off
Other process context (PC2)	On	Off	On	Off
P (Root Fault=True | PC1, PC2)	P_1	P_2	P_3	P_4
P (Root Fault=False | PC1, PC2)	1-P_1	1-P_2	1-P_3	1-P_4

Sometimes the impact of an operations process can be calculated when historical data is sparse. For example, if an operation calls a cloud API 12 times to manipulate resources within 8 minutes. During the operation, the occurrence probability of API failure increases and that can be calculated.

Acquiring probability is a challenging problem in practice. Acquiring probability from literature has the problem of coverage and context relevance. The empirical studies we referred to have different contexts and scopes. None of them can be regarded as comprehensive, in that it would cover all the possibilities and provide accurate distributions. However, precise probabilities are not necessarily required for accurate diagnosis, in our observation. The intuition is that certain operations processes significantly increase certain types of errors. As long as the relative ranking and the magnitude is reasonable, the diagnosis is relatively accurate. Adjustment to the probabilities can be made based on specific operational contexts.

D. Numerical Mapping

The numerical mapping derives a CPT for each of the intermediate fault and failure node, according to the type of gate present in the FT. Table II shows how the probability of intermediate faults with different logic gates are calculated from the parent set, again for the left-most branch in Fig. 3.

TABLE II: Probability calculation of the running example

FT Node	**FT Node Type**	**BN Probability (=True)**
Root1	Root fault	P_{r1}
Root2	Root fault	P_{r2}
Root3	Root fault	P_{r3}
Root4	Root fault	P_{r4}
Gate1	Intermediate fault (OR-gate)	P_{g1}=1-(1-P_{r2})×(1-P_{r3})×(1-P_{r4})
Gate5	Intermediate fault (AND-gate)	P_{g2}=P_{g1}×P_{r1}

V. Runtime Diagnosis

A. Belief Update Using Assertion Results as Evidence

As mentioned in Background section, BN supports different types of evidence. In our scenario, the diagnosis approach can receive both specific evidence and soft evidence.

1) Specific Evidence: Specific evidence refers to a definite finding that a node has particular value. When the diagnosis process is triggered, that triggering message contains the first sets of evidence: either a failed assertion or an anomaly detected by the monitoring mechanism, as well as (process) context information. All three can be considered definite in our current context, and thus considered specific evidence. In different contexts, this assumption can be dropped easily. The posterior probability or belief of the corresponding fault nodes (denoted as *Bel(Fault)*) are then either 0% or 100%. Once updated with specific evidence, *Bel(Fault)* is fixed, which cannot be changed any more.

2) Soft Evidence: Not all received evidence is definite. Evidence can be given as a probability distribution over the values of the corresponding node, which is known as soft evidence. Our approach assumes that assertions can have false positives/negatives in their results, and so the evidence gathered from them is taken as soft evidence.

Every fault, either root fault or intermediate fault, can be associated with multiple assertions. The occurrence of the fault is confirmed or ruled out through these on-demand assertions. For each assertion, we calculate the false positive and false negative rates from historical data. The reasons for inaccuracies in assertion results lie in the characteristics of the fault or the limitations of a certain assertion. For example, if the fault is transient, it is possible that by the time we do assertion checking, the fault already disappeared. To handle this issue, false positive/negative rates of assertions need to be considered with the evidence they present. Specifically, we denote an assertion's false positive rate as $P_{fp} > 0$ and false negative rate as $P_{fn} > 0$. When receiving a positive assertion result, the occurrence of the fault is confirmed with $(1 - P_{fp})$, and the occurrence of the fault is ruled out with $(1-P_{fn})$. After receiving an assertion result as a soft evidence, $Bel(Fault = T)$ is updated through applying Bayes' Theorem as shown below. Assume we have:

$P(Fault = T) = P_F, P(Fault = F) = 1 - P_F$ (prior probability)
$P(A = T|Fault = F) = P_{fp}$ (false positive rate of assertion)
$P(A = F|Fault = T) = P_{fn}$ (false negative rate of assertion)
$P(A = F|Fault = F) = 1 - P_{fp}$
$P(A = T|Fault = T) = 1 - P_{fn}$

where A represents $Assertion$. The belief in the fault occurrence can in general be calculated as:

$$
\begin{aligned}
&Bel(Fault = T) \\
&= P(Fault = T|A) \\
&= \frac{P(A|Fault = T)P(Fault = T)}{P(A)} \\
&= \frac{P(A|Fault = T)P(Fault = T)}{P(A|Fault = T)P(Fault = T) + P(A|Fault = F)P(Fault = F)}
\end{aligned}
$$

Once a positive assertion result is received, this changes to:

$$Bel(Fault = T) = \frac{(1-P_{fn})P_F}{(1-P_{fn})P_F+P_{fp}(1-P_F)}$$

If the assertion result is negative, the belief becomes:

$$Bel(Fault = T) = \frac{P_{fn}P_F}{P_{fn}P_F+(1-P_{fp})(1-P_F)}$$

After being updated with soft evidence, $Bel(Fault)$ is not fixed: it can be further updated when more soft evidence from more associated assertions is received. The question then is: what should be the threshold on probability to assume the occurrence of the fault? This is a configurable setting in our approach. Operators may e.g. feel confident to assume a fault occurred if $Bel(Fault = T)$ is higher than 90%.

3) Partial Belief Updates – Running Example: Table III shows how the beliefs are updated in the left-most branch of the running example based on different types of evidence.

TABLE III: Belief updates in the running example

Belief	Root1	Root2	Root3	Root4	Gate1	Gate5
No evidence	.0330	.0361	.0361	.0361	.1041	.0037
Failure=T	.6563	.2369	.2369	.2369	.6809	.6461
PC1=T	.8907	.3224	.3224	.3224	.9051	.8850
Gate1=T	.9778	.3562	.3562	.3562	1	.9778
Soft evidence of Root3	.9778	.3745	.3156	.3745	1	.9778
Root4=T	.9778	.0656	.0553	1	1	.9778
Root1=T	1	.0656	.0553	1	1	1

The first row shows the beliefs before receiving any evidence. Once the diagnosis is triggered by a failure, all the beliefs increase dramatically. Along with the evidence of failure, evidence of process context PC1 is also received, and the beliefs update results in all nodes within the left-most branch getting higher belief than the other branches. Among the six nodes, Gate1 has the highest belief, thus, we can check the assertion associated with Gate1 first. The received specific evidence confirms that Gate1 is true, which means Root2, Root3 or Root4 must have occurred. Since the three have the same belief, we randomly choose Root3 first. The corresponding assertion returns soft evidence against Root3, causing the belief of Root3 to be lowered. We then pick Root4 for the next check, and the returned assertion result confirms its definite occurrence. Since Gate1 is under an AND gate with Root1, we also check the assertion associated with Root1, and finally find out the root causes: Root1 and Root4.

B. Diagnosis Process

The BN is a graph $G(V, E)$. A subset of vertices represent the possible root causes, $R = \{v_i^r\}$, $R \subset V$. The goal of diagnosis is to identify the final set root causes $R^f \subset R$. The diagnosis process traverses over some or all nodes other than the process context nodes. Each vertex v_i is associated with a probability p_i, which changes through belief updating. v_i is also associated with assertions, a_{ij}, $1 \leq j$. Calling each a_{ij} returns either specific evidence in binary form (*true* or *false*), or soft evidence in form of a probability distribution over *true* and *false*.

Algorithm 1 captures the diagnosis process. When a diagnosis is triggered, it receives the values for context nodes and a failure node for the detected error as specific evidence. $Bel(PC)$ and $Bel(failure)$ are updated and fixed accordingly (line 1). $ToDo$ is the current to-check list of nodes, i.e., the assertions associated with nodes in this list need to be evaluated to find the (root) causes. $V \setminus PC$ is the set of intermediate fault nodes and (root) cause nodes in BN. Initially, all the intermediate fault nodes are added to $ToDo$ (line 2), which is updated as the diagnosis proceeds and more evidence is received.

The diagnosis stops when $ToDo$ is empty (line 3). While there are nodes left in $ToDo$, we firstly select a node v_i from $ToDo$ (line 4). The selection algorithm uses our online optimisation, as discussed in the next section. There are simple alternative selection methods – e.g., just selecting the node with the highest probability from $ToDo$ (as shown in the example of Table III), or selecting a node based on a predefined diagnosis runbook, as a human operator would usually do. We compare these three methods in our evaluation.

If v_i is a gate and if the belief of its parents (denoted as $parents(v_i)$) is fixed (line 5), $Bel(v_i)$ can be calculated from the belief of its parents (line 6). Thus, no assertions need to be checked and v_i is removed from $ToDo$ (line 7). Otherwise, an assertion associated with v_i is executed. Algorithm 1 doesn't consider any attributes of assertions, it selects an assertion associated with the node. $Bel(v_i)$ is updated based on the assertion result (lines 9-10). If the assertion result is specific evidence, $Bel(v_i)$ is fixed and v_i is removed from $ToDo$ (line 12); else, $Bel(v_i)$ may still be updated later on.

If the assertion result confirms the condition of v_i is true, lines 15-19 treat the case if v_i is a (root) cause node. There can be consecutive AND gates along the path from v_i to the failure node, which means v_i might not be the only root cause (lines 16-18). Thus, v_i is pointed to the furthest AND gate. Please note that BN doesn't have AND/OR gates, we keep the semantics of gates from FT. We use $child(v_i)$ function to refer to the child node of v_i. Since the BN is converted from a fault tree, every node on our BN has at most one child. At line 20, the (root) cause must be within the $subGraph(v_i)$, a sub graph of G defined as follows. $subGraph(v)$'s only leaf node is the current v from line 20, and root nodes in the subgraph are the (root) cause nodes that can be reached from v. Since the cause is confirmed to be in this subgraph, any node v_i' in $ToDo$ is removed from $ToDo$ if it is not in $subGraph(v_i)$ or if $Bel(v_i')$ cannot be changed anymore (line 22).

If the assertion result confirms the condition of v_i is false, it means the (root) cause is not within the $subGraph(v_i)$ (line

25). Similarly as before, if there are consecutive AND gates along the path from v_i to the failure node, it means the root cause is not in the subgraph with the furthest AND gate as the root (lines 26-28). Thus we iterate through $ToDo$ to remove the nodes with fixed belief, and the nodes that are contained in the subgraph (line 30). The belief of the nodes is updated before removing them from $ToDo$. After the diagnosis process stops, all the (root) cause nodes with $Bel(v = true) >= threshold$ are the root causes (Line 38).

C. Assertion Selection Optimisation

A critical step in the diagnosis process concerns which node from the to-check list to visit next. We present our approach for risk minimising selection in the following.

1) Attributes: We consider several attributes of both faults and associated assertions in our optimisation method. The **attributes of faults/failures** include occurrence probability, expressed by the belief of the corresponding node in the BN, and potential impact to the whole system, measured by partial capacity loss. The **attributes of assertions** include assertion execution duration and false positive / negative rates. Assertions have false positive and false negative rate because of the uncertainty during the measurement/asserting that affects accuracy. As mentioned earlier, false positive and false negative rates are accounted for in the BN through belief updating using soft evidence.

The potential partial availability loss is calculated by multiplying the duration of the assertion and the potential impact of the fault. Note that the total downtime includes detection, diagnosis, and recovery time. However, in the scope of this paper we can only *affect* diagnosis time, and hence target solely it in the optimisation. As mentioned earlier, there can be a strong need for human involvement to augment the automated assertions, which may take considerable time. For example, once the LC is asserted to be not consistent with what we expect after running automated checks, we may need a human operator to confirm that the current LC in use is indeed wrong, rather than a legitimate new one from a concurrent update.

2) Problem Analysis: Our target is to minimise the loss of capacity during the overall diagnosis process. Thus, the question is how to optimally select the next assertion(s) accordingly. Faults in our setting may have very different consequences. A fault in an individual VM may not have major consequences in a large-scale system. In contrast, a system-wide misconfiguration affecting all newly-added VMs may have major consequences.

According to the classic risk definition, e.g. [15], risk is calculated as the likelihood of an event multiplied with the potential consequence. To this end, each vertex v_i in G has an occurrence probability p_i and is assigned c_i^r, a real number denoting the relative capacity loss if the respective fault / root cause is present. A successful diagnosis process traverses a particular set of vertices, $S \subset V$. Note that there may be more than one root cause, i.e., multiple $\{v_i^r\}$ in S. Say, vertex v_k is being considered, and S_k is the set of nodes reachable from v_k, i.e., a path exists from v_k to any $v_i \in S_k$. We then define the maximal root cause capacity loss as $c_k^{max} = max(c_i^r | v_i^r \in S_k)$. At any point in time, our goal is to minimise the overall diagnosis time multiplied

Algorithm 1: diagnosis process

Input: $Evi(PC)$, $Evi(Failure)$
Output: R^f, the final root causes
Data: $G(V, E)$, the graph
Data: PC, the process context nodes
Data: R, the root cause nodes
Data: $ToDo$, the current to-check list
Data: a_{ij}, the j^{th} assertion associated with vertex v_i
Data: $Threshold$, the threshold probability used to decide root cause
Fn: $Evi(v)$ is the evidence received on v
Fn: $Bel(v)$ is the belief of v
Fn: $Parents(v)$ is the parent set of v
Fn: $Child(v)$ is the child of v

```
1  update Bel(PC), Bel(Failure)
2  ToDo ← V \ PC
3  while ToDo ≠ ∅ do
4      v_i, a_ij ← select vertex from ToDo and one of its assertions
5      if v_i is Gate && Bel(Parents(v_i)) is fixed then
6          update Bel(v_i)
7          ToDo ← ToDo \ {v_i}
8      else
9          Evi(v_i) ← run a_ij
10         update Bel(v_i)
11         if Evi(v_i) is specific evidence then
12             ToDo ← ToDo \ {v_i}
13         end
14         if Evi(v_i) == true then
15             if v_i ∈ R then
16                 while child(v_i) is an AND Gate do
17                     v_i = child(v_i)
18                 end
19             end
20             for each v'_i ∈ ToDo do
21                 if v'_i ∉ subGraph(v_i) || Bel(v'_i) is fixed
                   then
22                     ToDo ← ToDo \ {v'_i}
23                 end
24             end
25         else
26             while child(v_i) is an AND Gate do
27                 v_i = child(v_i)
28             end
29             for each v'_i ∈ ToDo do
30                 if v'_i ∈ subGraph(v_i) || Bel(v'_i) is fixed
                   then
31                     update Bel(v'_i = true) ← 0
32                     ToDo ← ToDo \ {v'_i}
33                 end
34             end
35         end
36     end
37 end
38 R^f ← {v_i | v_i ∈ R and Bel(v_i = true) >= Threshold}
39 return (R^f)
```

by the maximal capacity loss, denoted as the *consequence* $C = max_{v_k \in ToDo}(c_k^{max} \times \sum_{v_i \in S_k, \forall j} t_{ij})$, where t_{ij} is the time spent on evaluating assertion a_{ij}.

The challenge in solving this optimisation problem lies in the uncertain information before the next assertion(s) is selected. This is two-fold: (i) the posterior probabilities will be updated only after we have the evidence acquired by the assertion; (ii) the eventual (root) causes R^f are unknown before the end of the diagnosis. The closest problem in literature is the well-known *dynamic online shortest path* problem, in which the weights of edges are unknown (unweighted) or vary within a range [16]–[18]. This is similar to our optimization problem, because accumulating t_{ij} to a minimum along a successful diagnosis path is similar to find a shortest path. To the best of our knowledge, dynamic online shortest path is closest to our problem, but the algorithms to shortest path do not apply due to the following differences. First, in most online shortest path problems, there is no dependency among the weights, but in our problem there are strong dependencies. Second, to solve the shortest path problem it is sufficient to find such a shortest path, but in our problem even if $\sum_{v_i \in S, \forall j} t_{ij}$ is minimised we need to consider c_{max}. Third, R^f and c_{max} are known only after the diagnosis process completed. Fourth, a_{ij} cannot be simply deemed as a part of weight. Hence, on the one hand the existing algorithms for online shortest path do not apply for the optimal selection of next assertions, and on the other hand to know all posterior probabilities the BN requires $2^{|V|-1}$ updates (an assertion returns true or false, but the initial vertex does not need to be asserted). For these reasons we had to devise a novel online heuristic algorithm, which is described next.

3) Online Assertion Selection: Theoretically, one could check assertions on all potential (root) causes, in a random order, until a positive assertion result is found. Such a random walk would often lead to suboptimal results, due to failure impact and state space size. For a random walk, one can calculate an expected value for C. When $|R|$ is reasonably big, by introducing BN C should be much smaller than for random walk. Another naïve method is a brute force exhaustive search on G, but that would be even worse than a random walk, since $|V| > |R|$.

Instead, we propose a heuristic search approach, based on posterior probabilities and consequence C. Thus we introduce a new feature for each vertex in G, called *risk*:

$$K \quad = \quad C \times P$$

with C as defined above (maximal capacity loss for a diagnosis path) and P as the probability of finding the (root) cause on such a diagnosis path. It is not realistic to find the global optimum for either C and P. Therefore we propose online heuristic algorithm aiming at local optimum. In particular, we define K_{ij} to each assertion, that is, the risk associated with assertion a_{ij}. The basic idea of our online heuristic algorithm is to select next assertion with a limited clairvoyance on the future risk. To do this, we define d_m to be the maximum out-degree in G and d_p the depth to which the algorithm should look forward. Algorithm 2 basically computes the biggest K_{ij} from the candidate assertions for nodes in the to-check list.

Algorithm 2 is run for each vertex in the $ToDo$ list. For each run, the set of possible nodes with assertions within the clairvoyance is V^R. To cover each possible assignment of *true* or *false* to these assertions, we increment the integer $counter$ from 0 to $2^{|\mathbf{V^R}|}$ (lines 6, 7, and 18), and then interpret its value as binary array (line 8). The binary array represents one combination of the assertions in V^R being true or false. We assign the combination to a copy of the BN, which is updated by calling $Bel()$ (line 9). After the BN belief update, the posterior probabilities for the current combination of assertion results can be retrieved from the respective nodes. With these probabilities, we can compute the maximum risk for each path (lines 10-16). The risk of the path is the sum of assertion times multiplied with the probabilities of the nodes along the path, then multiplied with the maximal capacity loss of reachable root cause nodes (line 11). We store the first assertion on the path with the highest overall risk in a_{ij}^{max}, and ultimately return it as desired next assertion (line 18).

Algorithm 2: d_p-depth selection of the next assertion

```
   Input: current vertex v_c from ToDo list
   Output: next assertion a_ij
   Data: G(V, E)
   Data: the clairvoyance depth d_p to look forward
1  V^R ← v_i, ∀d(v_c, v_i) ≤ d_p − 1 and v_i ∉ R ; // all vertices
   within the distance d_p − 1
2  F^V ← v_i, ∀d(v_c, v_i) ≤ d_p or v_i ∈ R ; // front vertices, all
   vertices with the distance d_p
3  P^t ← all paths from v_c to v_i ∈ F^V ; // all paths from v_c
   to the front vertices within the clairvoyance
4  K^max ← 0;                          // maximal risk initiated to be 0
5  a_ij^max;                           // the assertion with the maximal risk
6  counter ← 0 ;        // count to the number of all possibilities
   within the clairvoyance
7  repeat
8      Convert counter into a binary and assign each bit
       to the sequence of v_i ∈ V^R;
9      Bel(V^R) ;                      // update posterior probabilities
10     for each path p_m^t ∈ P^t do
11         K^m ←
           c_u^max Σ_{v_i∈p_m^t, t_ij∈v_i, p_ij^a∈v_i} max(t_ij p_ij^a p_i), v_u ∈
           F^V ∩ p_m^t ;     // the maximum risk along a path p_m^t
12         if K^m ≥ K^max then
13             K^max ← K^m;
14             a_ij^max set to assertion with maximal
               consequence of the first vertex in p_m^t ;
15         end
16     end
17     counter ← counter + 1 ;
18 until counter = 2^{|V^R|+1};
19 return a_ij^max
```

From a given vertex out of the ToDo, this algorithm searches for the biggest future risk within the clairvoyance, which is scoped by d_p and implicitly d_m. We can see that the $2^{|V^R|}$ loop dominates the running time, and in each loop usually $Bel()$ takes the longest time. Hence the time complexity of this algorithm is $O(2^{|V^R|})$. In most cases, we hope that $|V^R|$ is a constant or bounded. In other words, we can assign a value α to $|V^R|$ and then calculate a reasonable d_p. $|V^R|$ can

be calculated from d_p and d_m. Thus, $\alpha = |V^R| = \sum_{i=1}^{d_p-1} d_m^i$. With some mathematical operations omitted here, we can set the depth $d_p = \lceil \frac{\lg(\alpha d_m + d_m - \alpha)}{\lg d_m} \rceil, d_m > 1, \alpha > 1$. There are some special cases. First, d_p cannot be greater than the diameter of G. When d_p is as big as the diameter of G, the online algorithm exhaustively searches over G. In this case, the time complexity is exactly $\Theta(2^{|V|-1})$. In the case of $d_m = 1$, all the faults are connected by a line and $d_p = \alpha + 1$. In practice, α can be a relatively big integer which implies a longer running time of this heuristic algorithm. Whether the gained accuracy worths waiting for better results depends on both the risk (because the optimization increases diagnosis time) and the duration of assertions: if they take tens of minutes, waiting a few more seconds for better selection results worth it. Furthermore, the assertion selection only changes if posterior probabilities in a related part of the BN change. Therefore, caching of the calculation can be done rather easily, and only for changed parts the optimisation needs to be updated.

VI. Experimental Evaluation

A. Real-world Scenarios

Through our engagement with industry, we accumulated a number of outage cases in cloud application operations. The faults and sequence events used come from these real world cases without implication or modification; so do Scenarios 1 and 3 below. Scenario 2 is a well-known problem reported by a number of Internet companies including Facebook, which we adapted to a cloud operation situation.

1) Uninstalled/disabled application software: An outage at Company A happened because newly replaced VMs in an auto-scaling group (ASG) did not have the main application properly installed. A misconfigured health check exacerbated the problem by replacing old healthy VMs with more misbehaving VMs. The root cause of this outage was incorrectly installed/ enabled application software in otherwise healthy VMs.

2) Mixed version: One of the most challenging errors during update is the mixed version error, which can be caused, for example, by changing a configuration during an ongoing upgrade. In a large-scale deployment, this can happen quite easily if different development teams push out changes independently during a relatively long upgrade process. This can result in mixed versions of the system co-existing.

3) Unauthorised access: An outage from Company B was caused by a developer's unauthorised access to the production database, or by mistaking the production database as test database if access is authorised to both.

B. Fault Injection

Based on the three real-world scenarios and past empirical study on Cloud developer forums and public industry outage reports [4], we selected a list of common and critical representative faults to be injected in the experiment, shown in Table IV. The empirical studies, especially on cloud operation issues, have descriptive statistics that guided us selecting the representative faults.

Two dimensions of faults are considered: scope and type of the fault. A fault can have a local scope, limited to an individual VM, or a group-wide scope that has impact on a collection of VMs. For the VM-level faults, we first select a VM from the ASG randomly, and then inject the fault into the selected VM. For the group-level faults, we directly inject the fault into the corresponding resource.

TABLE IV: Injected Faults

Fault	Type	Scenario
VM level		
F1. Uninstalled application software	Human operation error/ Network	Uninstalled/Disabled software
F2. Disabled application server	Software failure	Uninstalled/Disabled software
F3. Wrong setting of SG	Human configuration error	Unauthorized access
F4. Stuck launching VM	Cloud API failure	Developer forum
F5. Unavailable VM	Resource failure	Developer forum
Group level		
F6. Wrong SG used in LC	Human configuration error	Mixed version
F7. Wrong AMI in LC	Human configuration error	Mixed version
F8. Unavailable ELB	Resource failure	Industry outage report
F9. ELB config. error	Human configuration error	Developer forum

C. Experiment Methodology

We injected the 9 faults in an application running on AWS, in separate runs. As discussed earlier, faults have different occurrence probability in different process contexts. Most of the faults can occur during both system upgrade and normal operation. One set of exceptions are faults where the LC has wrong settings. Usually these faults would occur during system upgrade. But in the cloud, there are always operations like legitimate scaling in/out or replacing a VM due to failed health checks. Thus, these faults can still occur outside upgrades, but with an occurrence probability that is a lot lower.

For each of the 9 faults, we conducted 10 runs (i.e., 90 runs in total). Half of the runs are conducted under system upgrade, and the other half are conducted under normal operation. We compare three different kinds of selection methods: greedy probability-based selection, selection based on a pre-specified order, i.e., an error diagnosis *runbook*, and optimisation-based selection as per Section V-C. For each run of experiment, we use all three alternative assertion selection methods in parallel, from separate environments without interference. Thus, each of the three alternative selection methods was used on all 90 runs for diagnosis. We report the accuracy, efficiency and risk of alternative selection methods.

D. Experiment Setup

We used a cluster with 8 VMs deployed in AWS as an ASG. Netflix Asgard is chosen to assist the system upgrades of an ASG-based cluster. When the ASG finds an unhealthy VM (e.g. caused by termination), it replaces the VM with a new one. Asgard does rolling upgrade by changing the LC, and then terminating VMs from the ASG, utilizing the ASG to launch new VMs with the new version. The application on which the rolling upgrade is performed is a distributed log monitoring system running on the Ubuntu operating system.

An ASG starts new VMs according to its launch configuration (LC). A faulty field in the LC could affect all the VMs within the group. The VMs in the ASG are registered with an Elastic Load Balancer (ELB), which provides a single

point of contact for incoming traffic. Thus, if the ELB becomes unavailable or is misconfigured, it will affect the whole ASG.

VMs within the ASG are associated with a security group (SG). SG is used to separate testing environment and production environment. A faulty rule within the SG could allow a resource in the testing environment to access the production environment. The VMs are associated with a resource-based IAM policy as well to define the permissions of users based on organisational groups. The resource-based IAM policy works together with user-based one to restrict users' AWS access based on their roles, such as developer or tester. Furthermore, access to VMs is based on an ssh key pair. Upgrading a VM can include changing AMI, SG rules, key pair, IAM policy or other configurations, such as VM type or kernel ID.

E. Experiment Results

1) Accuracy: The diagnosis with all the three methods of assertion selection achieved 100% accuracy if the injected fault was not transient, meaning the root cause remained observable until a diagnosis finished. However, transient faults caused false negative cases in our diagnosis and thus, decreased the accuracy. Transient faults are common in cloud applications due to the sophisticated fault tolerance mechanisms at various levels. Whether a transient fault is detected depends on two factors: diagnosis time and the assertion used to find the fault. For example, ASGs can be used for fault tolerance. If a VM becomes unavailable, the ASG will start a new VM to replace the unavailable VM. Depending on what information is required for diagnosis, the original fault may get lost due to the replacement. Undiagnosed faults may come back.

2) Efficiency: Fig. 5 shows the diagnosis time of all the three selection methods over the 90 runs (270 data points). The y-axis represents diagnosis time; the x-axis is the index of the experiment with different faults being injected. For example, for x ∈ [1,10] the 30 times are plotted for diagnosing fault F1, as defined in Table IV. Diagnosis using greedy probability-based selection took between 0.97s and 2570.57s, with an average of 781.25s. Using selection based on pre-defined order took between 3351.35s and 9774.32s, averaging at 6903.10s, and optimisation-based selection between 480.20s and 2018.14s, with an average of 988.04s. The relative long diagnosis times are due to human involvement in some of the steps. If human steps took less time, the results would be proportionally similar. Among the diagnosis using probability-based and optimisation-based selection, F2 requires the longest diagnosis path, containing 10 steps under optimisation-based selection. F5 has the shortest path using probability-based selection, consisting of only 3 steps.

As can be seen, pre-defined order-based selection results in much slower diagnosis for all cases. For here on, we thus focus on the interesting comparison between greedy and optimisation-based selection. For this comparison, one should bare in mind that optimisation-based selection minimizes *risk*, not diagnosis time. Still, in the 30 runs for faults F2, F8, and F9, diagnosis time with optimisation-based selection is shorter than with probability-based selection. In contrast, for the 20 runs for F1 and F5, the greedy selection is much faster. The reason for the latter case is that the diagnosis with greedy probability-based selection of these two faults are fully automated, while optimisation-based selection picks several assertions involving humans. This is done because the observed fault may be a symptom of a severe error, which is ruled out first by the assertions involving humans, before automated assertions find the real, less severe fault.

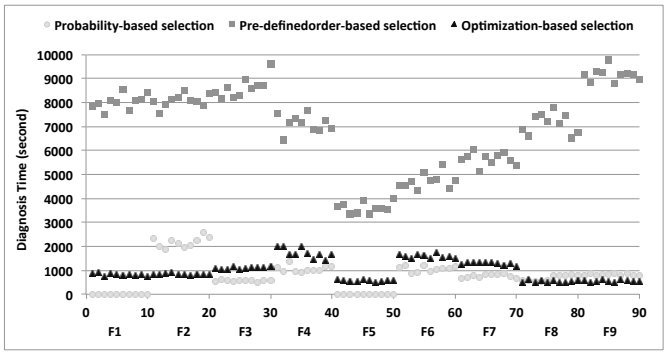

Fig. 5: Diagnosis time of the 90 runs.

3) Risk: To compare the risk during diagnosis with greedy and optimisation-based assertion selection, we calculated the ratio between the risk associated with the final diagnosis path of these two ways of selection. Say, the risk of final diagnosis path p_o of optimisation-based assertion selection o is $K(o, p_o)$ and that of greedy selection is $K(g, p_g)$, then we calculate the path risk ratio as $prr = \frac{K(g,p_g)}{K(o,p_o)}$. Not that, in order to *minimize the risk for the system*, we want to *maximize the risk on the diagnosis path*, i.e., take the path that could have the worst effects. Optimisation-based selection is thus beneficial where $prr < 1$. Fig. 6 shows prr for all 90 runs. In most cases (7 out of 9 faults) $prr < 1$, i.e., optimisation-based selection checked higher-risk paths than probability-based selection. For F1-F5, $prr < 0.1$, i.e., the risk on the diagnostic path of optimisation-based selection is much higher.

The two exceptions are F8 and F9, where $0.9 < prr < 8$. The final diagnosis paths of these two faults are totally different. Optimisation-based selection always starts with assertions with higher consequence and longer time. Thus, even though the ELB-related faults have a relatively low belief at the beginning of the diagnosis, optimisation-based selection checks them rather early, as these group-level faults result in maximal impact. In contrast, the greedy selection only considers the belief of the assertions, thus, ELB-related faults are checked only after several other assertions ruled out other faults. Through this ruling-out, the belief of ELB-related faults is increased to a very high value, and so is the corresponding risk at that (late) time. This is strongly reflected in $K(g, p_g)$. It should be mentioned, though, that albeit an unfavourable prr for F8 and F9, the actual diagnosis time is short using optimisation-based selection, and so the high impact of ELB unavailability or misconfiguration is actually reduced.

4) Diagnosis path discussion: In this section, we discuss the diagnosis path selected for three faults which are closely related to the three real-world scenarios introduced in Seciton VI-A. First we look at the diagnosis path for fault "F1. Uninstalled application software" when injected after system upgrade. The diagnosis is triggered by an increasing aggregated CPU utilisation after system upgrade. After the diagnosis starts, the checking starts from the latest launched VMs. The VM IDs of the latest launched VMs is achieved

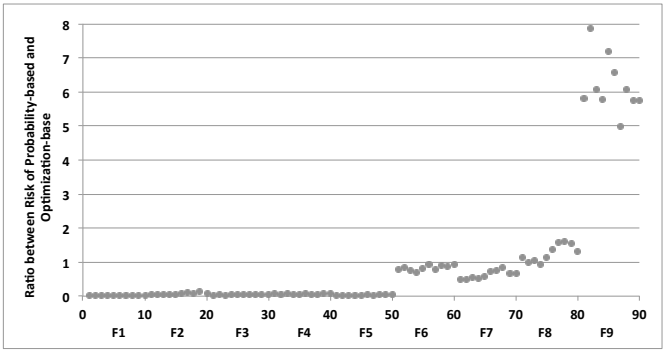

Fig. 6: Ratio between probability-based selection and optimisation-based selection.

by the first automated assertion costing around 3s. Then the statuses of the VMs are checked to identify the abnormal VM, which is an automated assertion, costing around 2s. After excluding the VM-level failure, the scope of the error is narrowed down to application-level fault that causes the failure of ELB VM health checking. After automatically checking the application-level health, around 0.3s, we confirmed that the fault is at VM-level. There are many faults that could cause ELB VM health checking to fail, such as ELB configuration error, authentication problem, an application software is not installed or enabled among others. Here fault F1 has the highest risk due to its higher probability caused by human operation error, network connection error or code repository down etc. Thus, the corresponding automated assertion is the first one to be checked at this level. Overall, the diagnosis path of this fault includes 4n assertion checking, n is the number of latest launched VMs. The average diagnosis time is 4.15s for one VM. The risk of optimisation-based diagnosis is 364 times of the risk of probability-based diagnosis on average.

The second one is the diagnosis path of fault "F7. Wrong AMI in launch configuration" injected during system upgrade. After checking several assertions, we found out that an VM fails the application-level health check. The fault at the next level with the highest risk is that "a VM is using an unexpected AMI". The VM is in an ASG (please refer to Fig. 3 for the relevant sub graph), Thus, one possibility is that the AMI configuration of launch configuration is not correct. It is also possible that the faulty VM is launched outside ASG with a wrong AMI, and attached to ASG. During system upgrade, the far left branch is traversed first because LC is much more possible to have problem and has larger consequence during system upgrade. The average diagnosis time is 791s. If the error is caused by a faulty VM being launched outside ASG and attached back to ASG, the confirmation of the fault might have up to 15 minutes delay for us to check the Cloudtrail[3] record to find when the VM is attached to the ASG by whom. The risk of optimisation-based diagnosis is 2 times of the risk of probability-based diagnosis on average.

The third one is the unauthorised access problem caused by fault "F3. Wrong setting of SG". Similar to the diagnosis of the second fault, we first found out that a certain VM fails the application-level health check. The access problem is the fault at the next level with the second highest risk. To confirm the access problem, the access log of the database deployed on the VM is checked manually, which may cost around 4 minutes. There are two faults that could cause the access problem, including wrong setting of SG and wrong IAM-policy setting. The first possibility is checked first and confirmed the root cause. To check the SG setting, we first automatically compare the current setting with a desired setting, and then manually go through the setting see if the setting is really correct. The average diagnosis time of this fault is around 9 minutes. On average, The risk of optimisation-based diagnosis is 36 times of the risk of probability-based diagnosis.

[3]http://aws.amazon.com/cloudtrail/

VII. Discussion and Limitation

Variation in Diagnosis Time: The diagnosis time of a fault is dominated heavily by the execution time of the assertions on the diagnosis path. The assertions' running times have a large variation. Automated assertions, e.g., those that can be checked by calling cloud API(s), finish within seconds. However, there are assertions that need to check information only becomes available after some delay – e.g., CloudTrail logs, which shows up to 15 minutes delay. There are other assertions that are hard to automate, e.g., checking several logs or checking complicated configurations. And time spent on assertions that involve human interaction is more varied, among others because it depends on the operator's capability.

Multiple Faults: If multiple faults must occur together to cause a failure, our diagnosis continues until all the required faults are identified. However, if not all the root faults are required to cause the failure, the current diagnosis method stops after identifying one of the faults causing the failure. We plan to extend the diagnosis algorithm for these cases.

Overlapping Faults: Some root faults are overlapping since we consider faults with differing scopes. Several group-level faults cause several VMs to turn into erroneous states. Thus, if we checked for VM-level faults, we could miss a group-level cause if we stopped after one VM-level fault is identified. Optimisation-based diagnosis avoids this situation because it always check for faults with higher consequence first.

Limitations: There are some obvious limitations of our approach. First, the fault tree still needs to be constructed and converted to BN manually and new faults need to be added. To make this approach applicable for systems with many different types of faults, automation in these aspects is needed. For cloud operations, we observed the number of faults to be limited.

The approach primarily applies to non-transient faults. Transient faults can only be diagnosed if the duration during which they are observable is longer than the time until diagnosis checks for them. Also, the approach relies on the completeness of the models – faults that are not modelled, or where the assertions are unable to find the fault, will not be found. The perfect diagnosis accuracy we observed in our experiments can only be achieved with complete models.

Acquiring probabilities is always difficult, especially when specific to a single organisation or operation. We believe there are commonalities among the different types of operations, reflected in their frequency of using the cloud APIs for infrastructure and configurations. Also, as argued above, exact probabilities are not a requirement for successful diagnosis.

VIII. Related Work

Inference in Bayesian Network: The basic task for a probabilistic inference system, like a BN, is to calculate the posterior probability distribution of a set of query nodes, given values of some evidence nodes. BN supports different directions of reasoning, including diagnostic reasoning, and predictive reasoning. Using BN to do *predictive reasoning* from new evidence about causes to new beliefs about effects has been used in reliability community in the past decade to analyse dependable systems [8], [11], [12]. In these work, BN works as an alternative of FT. We use BN in runtime error diagnosis of cloud operations. BN can be used to perform *diagnostic reasoning* from symptoms to causes. [19] argues that probabilistic or causal inference, like in BNs, is not suitable for cases where the values of the variables are multi-dimensional and real value-based data. BN-like inference graph-based diagnosis has been used largely for network problem diagnosis and simple node failures [20]–[22]. Our approach deals with API failures, resource failures confirmed by assertions with binary results, and categorical process context, where BN is an appropriate technique to use.

Error Diagnosis during Normal Operation: There are approaches based on statistics, machine learning, and rules for diagnosing errors during normal operation [23]–[27]. They are built for use during "normal" operations only, assuming a system has normal operation profiles that can be learned from historical data and deviations from the profiles can help detect, localise and identify faults. These assumptions do not hold in cloud operations. Our BN is currently constructed manually, but given enough historical data it can be learned.

IX. Conclusion

We proposed an error diagnosis approach using a Bayesian Network, which captures the occurrence probability of possible faults, errors, and failures under various contexts. Our diagnosis algorithm traverses the BN to perform diagnostic assertion checks associated with the nodes, and updates the belief based on the results. Which assertion to check next is selected through an online optimisation approach that minimises the risk of the diagnosis path. In our experiments, the diagnosis approach achieved perfect accuracy for non-transient faults. The presence of transient faults which disappear before diagnosis checks for them would reduce the accuracy. Our diagnosis approach can minimise the risk of most of the injected faults significantly. While we built our approach for and applied it to cloud application operations, we believe the range of possible applications to be much wider than that. In the near-term future we will therefore consider other applications, as well as focusing on performance tuning and possible higher prioritization of potentially transient faults.

Acknowledgment

NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program.

References

[1] D. Yuan, Y. Luo, and X. Zhuang et al, "Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems," in *OSDI*, 2014.

[2] H. S. Gunawi, M. Hao, T. Leesatapornwongsa, T. Patana-anake, and T. Do et al, "What bugs live in the cloud? A study of 3000+ issues in cloud systems," in *ACM SoCC*, 2014.

[3] O. Crameri, N. Knezevic, D. Kostic, R. Bianchini, and W. Zwaenepoel, "Staged deployment in mirage, an integrated software upgrade testing and distribution system," in *SOSP*, 2007.

[4] Q. Lu, L. Zhu, L. Bass, X. Xu, Z. Li, and H. Wada, "Cloud API issues: An empirical study and impact," in *QoSA*, 2013.

[5] Q. Lu, X. Xu, L. Bass, L. Zhu, and W. Zhang, "Mechanisms for tail-tolerant cloud operations for internet-based software," *IEEE Software*, 2014.

[6] X. Xu, L. Zhu, I. Weber, L. Bass, and D. Sun, "POD-Diagnosis: Error diagnosis of sporadic operations on cloud applications," in *DSN*, 2014.

[7] K. B. Korb and A. E. Nicholson, *Bayesian Artificial Intelligence (2nd Edition)*. CRC Press, 2010.

[8] N. Khakzad, F. Khan, and P. Amyotte, "Safety analysis in process facilities: Comparison of fault tree and bayesian network approaches," *Reliability Engineering and System Safety*, vol. 96, 2011.

[9] X. Xu, I. Weber, H. Wada, L. Bass, L. Zhu, and S. Teng, "Detecting cloud provisioning errors using an annotated process model," in *MW4NextGen*, 2013.

[10] A. Bobbio, L. Portinale, M. Minichino, and E. Ciancamerla, "Improving the analysis of dependable systems by mapping fault trees into bayesian networks," *Reliability Engineering and System Safety*, vol. 71, 2001.

[11] C. Hobbs, "Fault tree analysis with bayesian belief networks for safety-critical software," QNX Software Systems, Tech. Rep., 2010.

[12] S. Montani, L. Portinale, A. Bobbio, and D. Codetta-Raiteri, "RADYBAN: A tool for reliability analysis of dynamic fault trees through conversion into dynamic bayesian networks," *Reliability Engineering and System Safety*, vol. 93, 2008.

[13] K. Nagaraja, F. Oliveira, R. Bianchini, R. P. Martin, and T. D. Nguyen, "Understanding and dealing with operator mistakes in internet services," in *OSDI*, 2004.

[14] B. Wesely, "Fault tree analysis (FTA): Concepts and applications," NASA HQ, Tech. Rep., 2002.

[15] B. Boehm, Ed., *Software Risk Management*. IEEE Press, 1989.

[16] H. N. Djidjev, G. E. Pantziou, and C. D. Zaroliagis, "On-line and dynamic algorithms for shortest path problems," in *STACS*, 1995.

[17] A. Gyrgy, T. Linder, and G. Ottucsk, "The shortest path problem under partial monitoring," in *COLT*, 2006.

[18] E. Takimoto and M. K. Warmuth, "Path kernels and multiplicative updates," *Journal of Machine Learning Research*, vol. 4, 2003.

[19] S. Kandula, R. Mahajan, P. Verkaik, S. Agarwal, J. Padhye, and P. Bahl, "Detailed diagnosis in enterprise networks," in *ACM SIGCOMM*, 2009.

[20] P. Bahl, R. Chandra, A. Greenberg, S. Kandula, D. A. Maltz, and M. Zhang, "Towards highly reliable enterprise network services via inference of multi-level dependencies," in *ACM SIGCOMM*, 2007.

[21] X. Chen, M. Zhang, Z. M. Mao, and P. Bahl, "Automating network application dependency discovery: experiences, limitations, and new solutions," in *OSDI*, 2008.

[22] B. P. III, P. Ning, and S. Jajodia, "On the accurate identification of network service dependencies in distributed systems," in *LISA*, 2012.

[23] X. Pan, J. Tan, S. Kavulya, R. Gandhi, and P. Narasimhan, "Ganesha: blackbox diagnosis of mapreduce systems," *SIGMETRICS Perform. Eval. Rev.*, vol. 37, 2010.

[24] J. Tan et al, "Kahuna: Problem diagnosis for mapreduce-based cloud computing environments," in *NOMS2012*, 2010.

[25] S. P. Kavulya, S. Daniels, K. Joshi, M. Hiltunen, R. Gandhi, and P. Narasimhan, "Draco: Statistical diagnosis of chronic problems in large distributed systems," in *DSN*, 2012.

[26] H. Nguyen, Z. Shen, Y. Tan, and X. Gu, "FChain: Toward black-box online fault localization for cloud systems," in *ICDCS*, 2013.

[27] B. Sharma, P. Jayachandran, A. Verma, and C. R. Das, "CloudPD: Problem determination and diagnosis in shared dynamic clouds," in *DSN*, 2013.