Runtime Recovery Actions Selection for Sporadic Operations on Cloud

Min Fu1,2, Liming Zhu1,2, Daniel Sun1,2, Anna Liu1,2, Len Bass1,2, Qinghua Lu1
Software Systems Research Group, NICTA, Sydney, Australia1
School of Computer Science and Engineering, University of New South Wales, Sydney, Australia2

Abstract—Sporadic operations such as rolling upgrade or machine instance redeployment are prone to unpredictable failures in the cloud largely due to the inherent high variability nature of cloud. Previous dependability research has established several recovery methods for cloud failures. In this paper, we first propose eight recovery patterns for sporadic operations. We then present the filtering process which filters applicable recovery patterns for a given operational step. We also propose a methodology to evaluate the recovery actions generated for the applicable recovery patterns based on the metrics of Recovery Time, Recovery Cost and Recovery Impact. This quantitative evaluation will lead to selection of optimal recovery actions. We implement a recovery service and illustrate its applicability by recovering from errors occurring in Asgard rolling upgrade operation on cloud. The experimental results show that the recovery service enhances automated recovery from operational failures by selecting the optimal recovery actions.

Keywords—recovery oriented computing; infrastructure operations and automation; reliable cloud based systems;

I. INTRODUCTION

Sporadic operations on cloud mean the deployment or maintenance operations which are relatively less frequent and regular than normal activities like transactions in an e-commerce application. Automatic recovery from failures during sporadic operations on cloud such as rolling upgrade or machine instance (virtual machine) redeployment has become increasingly important because of the need to manage the uncertainty on cloud and the highly variable SLA (Service Level Agreement) of cloud based systems[1]. Further, as the practice of DevOps (Development & Operations) becomes more prevalent, it is commonplace that modern IT departments make frequent system builds and deployments on a daily basis[1]. Continuous deployments of automated systems cannot afford downtime and this means that manual recovery from errors occurring in operational processes is not an acceptable solution[1].

One virtue of sporadic operations is that every execution of an operation with different applications (the data to the operation) performs the same set of steps[2]. Rolling upgrade[3], for example, removes one instance from service and replaces it with an upgraded instance regardless of the application being upgraded. This regularity allows the creation of the process model of an operation[4]. We are interested in recovering from failures that occur during correctly specified operations on cloud. These failures are not due to incorrect specification (e.g., incorrect configurations or incorrect operation workflow) but are due to the uncertainty and highly variable performance and availability of operations executed in the cloud[5] or from race conditions caused by simultaneous manipulations on the resources required by the sporadic operation.

In this paper, we first propose eight recovery patterns which represent the logical recovery flows for an error: 1) Compensated Undo & Redo; 2) Compensated Undo & Alternative; 3) Rewind & Replay; 4) Rewind & Alternative; 5) Reparation; 6) Direct Redo; 7) Direct Alternative; and 8) Farther Undo & Redo. One thing to note is that among all eight recovery patterns not every pattern is feasible. For example, for the operation of deleting a data drive, Rewind & Replay or Compensated Undo & Redo are not feasible because the resource deletion operation cannot be rewound or undone. Similarly, the operation of “assigning IP address” is a process largely controlled by the cloud service provider[6], hence the recovery pattern of Direct Redo is not feasible. Part of our solution here is that we have built up a knowledge base of these infeasible actions in the cloud operations context, which can then be used to filter out the non-applicable recovery patterns. After reducing to the applicable recovery patterns set, we generate the recovery action for each applicable recovery pattern, and each recovery action consists of a list of relevant cloud APIs that represent the workflow of its recovery pattern.

We also propose three important metrics for evaluating these recovery actions: Recovery Time, Recovery Cost, and Recovery Impact. Recovery Time refers to the time taken for a recovery action’s workflow to recover from an erroneous or failed state to a correct or desirable state. Recovery Cost is defined as all monetary cost incurred by executing the recovery action. This dynamic monetary measurement is particularly important in the context of “pay-per-use” clouds[6]. Recovery Impact means the negative performance impact on the cloud system caused by the recovery action. One thing to note is that each of these metrics is meaningful to the business and hence the business can specify recovery requirements in terms of these metrics that the operational recovery must satisfy. The business may specify the requirements by setting an allowed boundary value for each of the three metrics.

The generated recovery actions within the applicable recovery patterns are not all guaranteed to be able to satisfy the recovery requirements. For example, the recovery action requiring terminating a cloud instance (virtual machine) may
need a long execution time that exceeds the maximum time duration boundary allowed by the business stakeholders. Hence, one challenge of operational recovery is its ability to satisfy the recovery requirements. We achieve this by using a Pareto set searching algorithm to search for the optimal recovery actions based on the predicted values of their recovery evaluation metrics.

We constructed a prototype recovery service with the following capabilities: 1) it filters the applicable recovery patterns; 2) it computes the values of recovery evaluation metrics for the generated recovery actions within applicable recovery patterns; and 3) it selects an optimal recovery action for execution at runtime if failures occur. We implement the prototype of the recovery service in a non-intrusive way (without modifying the source code of the operations) and evaluate its performance in an Asgard-based[7] rolling upgrade case study. Asgard is a cloud management tool developed by Netflix[7]. Rolling upgrade is a strategic upgrade method for cloud systems aimed at maintaining their service during upgrading. The experimental results show that the prototype recovery service can select the optimal recovery actions that best satisfy recovery requirements.

The contributions of this paper are: 1) we propose and evaluate a novel non-intrusive recovery framework for sporadic operations on cloud. Essential to our methodology are the applicable recovery patterns filtering and the selection of an optimal recovery action that can best fulfill the recovery requirements based on the three recovery evaluation metrics proposed by us; 2) we propose three recovery evaluation metrics and show how to calculate them; 3) we propose a Pareto set searching algorithm to solve our recovery actions selection problem.

The rest of this paper is organized as follows: section II presents the background; section III describes the eight recovery patterns; section IV describes recovery evaluation metrics; Section V presents our prototype recovery service; section VI describes the experiment and evaluation; section VII discusses threats to validity; section VIII is related work; and section IX provides the conclusion and our future work.

II. BACKGROUND

A. Cloud Sporadic Operations

Example Sporadic operations on cloud include installation, upgrade and reconfiguration. Here we use rolling upgrade as the illustrating example. In a rolling upgrade, a subset of instances currently running an old version of a software system are taken out of service and replaced with the same number of instances running a new version of the software system[8]. Rolling upgrade is the industry standard technique for moving to a new version of software that runs across a large set of servers[8]. One of the industry defacto standard tools used to perform rolling upgrade is Asgard[7], which is a cloud management tool specifically for Amazon Web Services (AWS) Elastic Compute Cloud (EC2)[6]. The procedure of Asgard rolling upgrade is illustrated in Fig. 1. It is derived using a process mining technique with Asgard execution logs as input data and Asgard source code analysis[4]. We can see that the rolling upgrade operation consists of 7 steps, where step 1 to step 3 are sequential, and step 4 to step 7 are iterative. In step 1, new LC (Launch Configuration) pointing to new AMI (Amazon Machine Image) is created; in step 2, the existing ASG (Auto Scaling Group) is reattached to the new LC (Launch Configuration); in step 3, the rolling policy (including instance killing order and killing number) specified by user is set; from step 4 to step 7, the system removes an old instance (virtual machine) from ELB (Elastic Load Balancer) and terminates it, then it relies on ASG (Auto Scaling Group) to launch a new instance (virtual machine) and registers the new instance (virtual machine) in ELB (Elastic Load Balancer). Steps 4 to 7 are iteratively executed until all the old instances are upgraded.

![Fig. 1. Asgard Rolling Upgrade Operation.](image)

B. Errors and Failures during Sporadic Operations

Due to the uncertainty and highly variable performance characteristic of cloud API functions[5], the sporadic operations such as rolling upgrade which rely on those API functions could fail at unpredictable times. For example, launching a new cloud instance relies on the cloud API of “RunInstances”, and it could fail because the API totally does not work and no instance is successfully launched, or the instance launching time is too long. These failures result from the instability of cloud and hence are inevitable. Some existing empirical studies[9] presented the failure rates of some cloud API functions, as shown in Table I.

<table>
<thead>
<tr>
<th>Cloud API</th>
<th>Failure Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. RunInstances</td>
<td>3.1%</td>
</tr>
<tr>
<td>2. TerminateInstances</td>
<td>3.9%</td>
</tr>
<tr>
<td>3. StartInstances</td>
<td>1.9%</td>
</tr>
<tr>
<td>4. StopInstances</td>
<td>1.8%</td>
</tr>
<tr>
<td>5. AttachVolume</td>
<td>0.3%</td>
</tr>
<tr>
<td>6. DetachVolume</td>
<td>3.2%</td>
</tr>
<tr>
<td>7. RegisterInstancesWithLoadBalancer</td>
<td>1.5%</td>
</tr>
<tr>
<td>8. DeregisterInstancesFromLoadBalancer</td>
<td>1.5%</td>
</tr>
</tbody>
</table>
C. Operation Errors Detection

In the work related to error detection and diagnosis for sporadic operations on AWS cloud[4], the authors proposed and implemented an error detection framework named POD-Diagnosis[4]. POD-Diagnosis is a model-based approach that explicitly models a sporadic operation as a process, uses the process context to locate errors, filters logs, visits fault trees, and performs on-demand assertions evaluation for online error diagnosis and root cause analysis[4]. Once errors are located, POD-Diagnosis[4] will trigger our recovery service to recover from errors. When POD-Diagnosis triggers our recovery service, it will pass the following information to the recovery service as its inputs: 1) current erroneous system state; 2) the previously captured consistent system state; 3) the expected system states before and after that step; 4) the consistent system state before the last step prior to the current step; and 5) the current step specification. In order for our recovery service to compute the recovery impact, POD-Diagnosis also passes to our recovery service the cloud system’s current capacity information such as current workload and current instances number.

III. EIGHT RECOVERY PATTERNS

The eight recovery patterns we propose are inspired from the existing recovery mechanisms for long running transactions[11]. For long running transactions, recovery strategies usually involve backward recovery and forward recovery[11]. Backward recovery first reverts the current erroneous state to a previous correct state before attempting to continue execution[11]. Forward recovery attempts to correct the current erroneous state and then continues normal execution[11]. One form of forward recovery is compensation[11], which means to attempt to correct the state of a system given some knowledge of the previous actions of the system[11]. In the context of cloud, some activities might have alternatives which lead to the same execution results as the original activities, so re-executing an activity is equal to executing its alternative. For example, waiting for auto scaling group to launch a new instance is equal to attaching a pre-prepared new instance into the auto scaling group. This is one aspect that has not been considered by recovery mechanisms for long running transactions. Another opportunity in cloud recovery is that some steps can be recovered by simply re-executing them. As such, we have organized this body of knowledge and propose eight recovery patterns: 1) Compensated Undo & Redo; 2) Compensated Undo & Alternative; 3) Rewind & Replay; 4) Rewind & Alternative; 5) Reparation; 6) Direct Redo; 7) Direct Alternative; and 8) Farther Undo & Redo.

Fig. 2 illustrates the mechanisms of these eight recovery patterns. Suppose the error occurs when process step X (Step X) of the operation is actually running. After detecting errors, the recovery is triggered by the error detection service. Five items of system states are the inputs to the recovery: $S_{err}$, $S_1$, $S_2$, $C_1$, and $S_0$. $S_{err}$ is the erroneous state after Step $X$, $S_1$ is the expected state before Step $X$ and $S_2$ is the expected state after Step $X$; $C_1$ is the captured state before Step $X$. $S_0$ is the expected state before the last step prior to Step $X$ which is Step X-1. Importantly, a step could have its alternative which leads to the same running result as the step itself (alternative of Step X (Alternative X) has the same running result as Step X). Compensated Undo & Redo means to make the current erroneous state into the expected system state before the current step and then re-execute the current step ($S_{err} \rightarrow S_1 \rightarrow \text{Step} X \rightarrow S_2$). Compensated Undo & Alternative means to make the current erroneous state into the expected system state before the current step and then execute the alternative of the current step ($S_{err} \rightarrow S_1 \rightarrow \text{Alternative} X \rightarrow S_2$). Rewind & Replay means to make the current erroneous state into the consistent system checkpoint before the current step and then re-execute the current step ($S_{err} \rightarrow C_1 \rightarrow \text{Step} X \rightarrow S_2$). Rewind & Alternative means to make the current erroneous system state into the expected system state before the current step and then execute the alternative of the current step ($S_{err} \rightarrow C_1 \rightarrow \text{Alternative} X \rightarrow S_2$). Reparation means to directly make the current erroneous system state into the expected system state for the current step ($S_{err} \rightarrow S_2$). Direct Redo means to directly re-execute the current step ($S_{err} \rightarrow \text{Step} X \rightarrow S_2$). Direct Alternative means to directly execute the alternative of the current step ($S_{err} \rightarrow \text{Alternative} X \rightarrow S_2$). Farther Undo & Redo means to make the current erroneous system state into the expected system state before the last step prior to the current step and then re-execute the last step prior to the current step and then re-execute the current step ($S_{err} \rightarrow S_0 \rightarrow \text{Step} X-1 \rightarrow \text{Step} X \rightarrow S_2$). During any recovery action, the recovery itself might fail due to reasons such as cloud APIs uncertainty[5] and it makes the system go into another unexpected erroneous state ($S_{err}$). If this happens, the recovery for the recovery action itself will be guaranteed by our “recovery for recovery” mechanism, which is to retry the recovery action recursively.

![Eight Recovery Patterns](image)

IV. RECOVERY EVALUATION METRICS

As far as businesses are concerned, the time elapsed during the recovery process for their systems may directly affect their commercial interest[12]. For example, the downtime of E-commerce website due to error recovery may result in loss of tens of thousands of dollars. So Recovery Time (RT) is first of the metrics that we use to evaluate recovery actions. Businesses often operate with a specific
Recovery Time Objective (RTO)\textsuperscript{[13]} in mind. Modern cloud services are often charged on a “per-use” basis, hence another factor cloud consumers care about is how much money is spent on running operations on the cloud systems. In the cloud context, recovery actions such as launching a new instance will incur additional monetary cost\textsuperscript{[6]}. We define such monetary cost as recovery cost. Hence, Recovery Cost (RC) is another metric used for evaluating recovery actions. Cloud systems can be messed up largely due to inappropriate recovery actions performed on the system\textsuperscript{[14]}. In other words, some recovery actions can have negative impact on the cloud system. It was mentioned that a recovery service must have a minimal consequence on the performance of each application\textsuperscript{[13]}. Hence, reducing the recovery’s negative performance impact on the system is imperative. Therefore Recovery Impact (RI) is also a metric for evaluating recovery actions. The definitions of the three proposed recovery evaluation metrics are shown in Table II.

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recovery Time (RT)</td>
<td>The time for a step in a sporadic operation to recover from an error state to a correct state</td>
</tr>
<tr>
<td>Recovery Cost (RC)</td>
<td>All monetary cost incurred by all the cloud API functions included in the recovery action</td>
</tr>
<tr>
<td>Recovery Impact (RI)</td>
<td>The negative performance impact of recovery on the cloud system being operated on during the time period of doing recovery</td>
</tr>
</tbody>
</table>

1) Recovery Time (RT). For normal activities (such as application workflow) on cloud system, recovery time is defined as the time for a system to recover from a failure to an agreed service level\textsuperscript{[13]}. Sporadic operation recovery time is different from normal operation recovery time. In the context of sporadic operations, recovery time’s definition has two aspects: 1) the time for the full system (during sporadic operation) to return from failure to the potentially degraded SLA/capacity; and 2) the time for the operation returning from a failure (or erroneous state) to an accepted state (could be early captured consistent state, or future expected state or even a “safe” state to proceed further). When we measure the recovery time for sporadic operations, we actually measure the execution time of the recovery actions which are consisted of a set of relevant cloud APIs.

2) Recovery Cost (RC). It means the money charged by AWS during recovery for cloud APIs (e.g. RunInstances) which composite the recovery action. We can get this information from the pricing policies in AWS website\textsuperscript{[6]}. Actions such as “RunInstances” will cause additional monetary cost. Actions such as “ModifyInstanceType” may also introduce additional monetary cost. By summing up the money spent by each cloud API in the recovery action, we can obtain the monetary cost of that recovery action.

3) Recovery Impact (RI). It means the negative impact caused by the recovery actions on the cloud system during the recovery period. Recovery impact comes from three aspects: 1) recovery actions will have a delay on the original completion time of the operation. 2) Certain recovery actions (e.g. terminating instances) will impose a decrease on the capacity/SLA level of the system. 3) Sometimes, the false positives of error detection will trigger the recovery which is actually unexpected, and such case might mess up the cloud system as well. From system users’ perspective, when it comes to discussing over system performance, they are mostly concerned with the system’s average response time for the user requests. Hence, we use the system’s average response time to user requests during the recovery period to measure the recovery impact on the cloud system being operated on.

V. OUR PROTOTYPE RECOVERY SERVICE

Our recovery service is designed with the aim of satisfying the recovery requirements. The overview of our recovery service prototype is described in Fig. 3. Error recovery will be triggered after detecting the error for a certain step. Among the eight recovery patterns, maybe not all of them are applicable. Hence, we determine the applicable recovery patterns by performing state reachability checking, idempotence checking and step alternative existence checking. Each applicable recovery pattern could contain several recovery actions. Hence, after filtering the applicable recovery patterns we manually generate the recovery actions for each applicable recovery pattern. Certain recovery actions within applicable recovery patterns might not be acceptable because they may fail in satisfying recovery requirements defined by the business. We compute the values of the recovery metrics of each recovery action. The recovery metrics are Recovery Time, Recovery Cost and Recovery Impact. Finally, we select an acceptable recovery action which satisfies the recovery requirements specified by the business stakeholders and execute it. The recovery method is able to cater for different types of clouds, because the cloud properties and functionalities required by our recovery method are shared among all different types of clouds, i.e., all types of clouds have cloud APIs provided, have instances running on them, and incur monetary costs.

A. Applicable Recovery Patterns Determination

Not every recovery pattern is applicable. For example, the recovery pattern of Compensated Undo & Redo can be non-applicable because the previous state cannot be reachable from current state, or the recovery pattern of
Compensated Undo & Alternative can be non-applicable because there is no step alternative existing, or the recovery pattern of Direct Redo can be non-applicable because it is not valid to do recovery just by re-executing the step (in other words, the current step is not idempotent). Hence, to determine the applicable recovery patterns, we need state reachability checking, idempotence checking as well as step alternative existence checking.

We perform state reachability checking by looking at what operational actions involved in the state transferring in the context of sporadic operations on cloud, the state transition involving stateful data drive (or store) creation and IP address reassignment are not feasible. Creating a stateful data drive is infeasible because the original data inside the drive is lost forever and cannot be replicated, even though the empty drive can be created by calling relevant cloud API function. Reassigning a new IP address to a cloud instance is not feasible because this action by cloud consumers is not allowed by cloud platform due to the limited visibility and control of cloud[15]. Hence, as long as the state transition does not involve data drive creation or IP address reassignment, it is feasible. The state structure is consisted of limited number of cloud resource items, and the state reachability checking function loops through all the resource items to check if there is IP address reassignment or data drive creation involved. If no, the state is reachable.

Idempotence checking is performed by examining whether the system can be recovered by merely rerunning the current step. There are two scenarios that the recovery can be only rerunning the current step. The first scenario is that, if the current erroneous state is the same as the expected state or the captured consistent state prior to the step, we can just rerun the current step for recovery and in this case it is actually the same as Compensated Undo & Redo or Rewind & Replay. One of such cases is the step of launching a cloud instance, and if this step fails the recovery action can be just redoing this step. So in this scenario the idempotence checking is actually looking at whether the current erroneous state equals to the consistent state before the step or not. The second scenario is that, no matter what is the current erroneous state, recovery by rerunning of the step will always yield the same expected result. One such example is recovery for the step of updating auto scaling group with newly created launch configuration in rolling upgrade operation, and no matter what is the current state (e.g. auto scaling group is attached with another unknown launch configuration) after this step, if we recover by rerunning this step, the auto scaling group will always be attached with the expected new launch configuration.

Step alternative existence checking is performed by checking if there exists any alternative for a certain step. Some steps may have alternatives, for example, the step of launching a new instance in auto scaling group has an alternative which is attaching a new instance with auto scaling group. We provide a mapping list which provides the step alternatives for various cloud operational steps and we use this mapping list to check if a step has any alternatives given the specification of the step.

The logic of determining the applicable recovery patterns is as follows: first, error detection service passes to the recovery service seven items of information as the inputs: 1) the current erroneous state (S_err); 2) the expected state (S_i) before the current step; 3) the captured consistent state (C_i) before the current step; 4) the expected state (S_j) after the current step; 5) the specification of the current step (Step X); 6) the specification of the last step prior to the current step (Step X-1); 7) the expected state (S_k) before the last step prior to the current step. Second, we construct all the eight recovery patterns by using the 7 items of information passed by error detection service. Then, for each recovery pattern, we check if it is applicable. For Compensated Undo & Redo, we check if S_i is reachable from S_err, and if so Compensated Undo & Redo will be included in the applicable recovery patterns list. For Compensated Undo & Alternative, we check if S_i is reachable from S_err and if alternative of Step X exists, and if so Compensated Undo & Alternative will be included in the applicable recovery patterns list. For Rewind & Replay, we check if C_i is equal to S_j (because C_i might be different from S_j due to state capturing service delay or false positives of state capturing; if so, C_i is not valid and Rewind is invalid) and if C_i is reachable from S_err, and if so Rewind & Replay will be included in the applicable recovery patterns list. For Rewind & Alternative, we check if C_i is equal to S_i, if C_i is reachable from S_err and if alternative of Step X exists, and if so Rewind & Alternative will be included in the applicable recovery patterns list. For Reparation, we check if S_i is reachable from S_err, and if so Reparation will be included in the applicable recovery patterns list. For Direct Redo, we issue idempotence check to see if it is feasible to recover by rerunning Step X, and if so Direct Redo will be included in the applicable recovery patterns list. For Direct Alternative, we check if alternative for Step X exists and we perform idempotence check to see if it is feasible to recover by rerunning Step X, and if so Direct Alternative will be included in the applicable recovery patterns list. For Farther Undo & Redo, we check if S_i is reachable from S_err, and if so Farther Undo & Redo will be included in the applicable recovery patterns list. Now we have obtained the final list of applicable recovery patterns and it will be returned.

### B. Recovery Actions Candidates

In the context of cloud sporadic operations, generating the recovery action for each applicable recovery pattern is straightforward, so the recovery action is manually generated. The cloud operator just needs to determine what actions need to be performed in order to transit current erroneous system resource state into the goal state, and what operational steps or alternatives need to be executed. For example, the generated recovery action for the recovery pattern of Rewind & Replay for step 5 (Terminate old version image instance) in rolling upgrade is described in Table III. The error occurring is that the instance termination takes too long time. Each recovery action consists of several activities which each are mapped with relevant cloud APIs. For example, the activity of “Terminate old version instance” is mapped with the cloud API of “TerminateInstancesInAutoScalingGroup”.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Compensated Undo &amp; Redo</td>
<td></td>
<td></td>
<td>Compensated Undo &amp; Redo</td>
<td>S_i reachable from S_err</td>
<td>Direct Redo</td>
</tr>
<tr>
<td>Rewind &amp; Replay</td>
<td></td>
<td></td>
<td>Rewind &amp; Replay</td>
<td>S_i equal to S_j</td>
<td>Direct Redo</td>
</tr>
<tr>
<td>Rewind &amp; Alternative</td>
<td></td>
<td></td>
<td>Rewind &amp; Alternative</td>
<td>S_i reachable from S_err and alternative of Step X exists</td>
<td>Direct Redo</td>
</tr>
<tr>
<td>Reparation</td>
<td></td>
<td></td>
<td>Reparation</td>
<td>S_i is reachable from S_err</td>
<td>Direct Redo</td>
</tr>
<tr>
<td>Farther Undo &amp; Redo</td>
<td></td>
<td></td>
<td>Farther Undo &amp; Redo</td>
<td>S_i is reachable from S_err</td>
<td>Direct Redo</td>
</tr>
</tbody>
</table>
TABLE III. RECOVERY ACTION GENERATED FOR REWIND & REPLAY

<table>
<thead>
<tr>
<th>Process Step</th>
<th>Recovery Actions (Rewind &amp; Replay)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step 5: Terminate old version image instance (VM) in ASG (Error: Instance termination fails by taking too long time)</td>
<td>Terminate the old version image instance; Launch another old version image instance and attach it to ASG; Terminate this newly launched old version image instance;</td>
</tr>
</tbody>
</table>

C. Recovery Evaluation Metrics Calculation

The acceptable recovery actions will be selected by using the recovery evaluation metrics of Recovery Time (RT), Recovery Cost (RC) and Recovery Impact (RI). We illustrate how they can be measured.

The calculation of Recovery Time is through summing up the execution time of each cloud API that is composing the recovery action. Recovery actions will be consisted of a limited number of cloud APIs provided by cloud platform and we obtain the execution time of each cloud API by empirical study on these APIs[5]. For example, the execution time of the API “UpdateAutoScalingGroup” is about 3 seconds, and the execution time of the API “RunInstances” is about 60 seconds. In our recovery service there is a mapping list between cloud APIs and their execution time.

The calculation of Monetary Cost is done by summing up the monetary cost of each cloud API that composes the recovery action. For example, the API of “RunInstances” will cost $0.0031 if it is launching a t1.micro typed instance. Some APIs have parameters and these parameters might influence the monetary cost. For example, if we launch two t1.micro typed instances by calling “RunInstances” API, the monetary cost will become $0.0062. The monetary cost of each cloud API is obtained from the pricing list defined by cloud platform[6]. In our recovery service there is a mapping list of cloud APIs and their monetary costs.

The calculation of recovery impact during recovery is illustrated in Fig. 4. We use the average response time of system workload requests to represent the system capacity because response time is a big concern to system users. Suppose an operation consists of five steps and during each step there is corresponding response time. Due to the variability nature of cloud systems, we typically observe varying response time for each step, however to simplify things, we simply take the average response time value for each step for the purpose of summing up performance impact.

Fig. 4(a) shows the response time trajectory of a successful operation where there is no error occurring or recovery triggered. The operation is assumed to consist of five steps (step 1 to step 5), and during each step there is relevant response time. We define $S_n$ (1 <= n <= 5) as the area of the shape which is formed by each step’s response trajectory and the step’s execution time. The overall area $S = S_1 + S_2 + S_3 + S_4 + S_5 = S_1 + S_2 + S_3 + S_4 + S_5$. The impact of the recovery action is calculated to be $S' = S$.

The impact of this erroneous operation with recovery is $S'$ which is equal to $S_1 + S_2 + S_3 + S_4 + S_5$. The impact of the recovery action is calculated to be $S' - S$, which is equal to $S_5 - S$.

Fig. 4(c) shows the response time trajectory of a successful operation where there is no error in step 2 but error detection service wrongly reports the error and triggers the recovery (false positive). The overall impact of the operation in this case is computed to be $S''$, which is equal to $S_1 + S_2 + S_3 + S_4 + S_5$, and the impact of the recovery action in this case is calculated to be $S'' - S$, which equals to $S_5 - S$. If the probability of false positive in error detection is denoted by $Pr(fp)$, then the true positive rate denoted by $Pr(tp)$ is equal to 1 - $Pr(fp)$. Then the overall impact of recovery is calculated to be $(1 - Pr(fp)) \times (S_1' + S_2 + S_3 + S_4 + S_5' + Pr(fp) \times SR_2')$, which is equal to $(1 - Pr(fp)) \times (S_1' + S_2 + S_3) + (1 - Pr(fp)) \times SR_2' + Pr(fp) \times SR_2'$. For the error in a certain step, no matter what recovery action is taken, the impact of the error itself keeps the same. Hence, $S_2' - S_2$ will be the same value for the same step. Hence, the relative impact of recovery action can be further denoted by $(1 - Pr(fp)) \times SR_2' + Pr(fp) \times SR_2'$. Suppose the recovery action is comprised of n cloud APIs, then both $SR_2$ and $SR_2'$ are equal to $\sum W_i \times T_i$, where 1 <= i <= n and $W_i$ is the response time during the execution of the $i^{th}$ cloud API and $T_i$ is the execution time of $i^{th}$ cloud API. Hence, the recovery impact will be calculated as $(1 - Pr(fp)) \times (\sum W_i \times T_i) + Pr(fp) \times (\sum W_i \times T_i)$, which is equal to $\sum W_i \times T_i$, where 1 <= i <= n. This formula can be further represented as below:

$I (Impact) = W \times T$, where $W$ denotes the matrix of the system response time during each of the cloud APIs and $T$ denotes the matrix of the execution time of each of the cloud APIs.

![Fig. 4](image)

**Fig. 4. Recovery Action Impact Analysis.**
System response time \((W_i)\) depends on the application running in the cloud, and hence in order to obtain system response time we need to have the application’s specification and features. We use a typical cloud benchmarking application to represent the real cloud application. The benchmarking application is TPC-W[16], and it is a three-tier e-commerce website which is widely used to benchmark the performance of data centres[17]. We use its workload generator to generate read requests and observe the average response time of those requests. The obtained response time represents the capacity of the cloud system during the operation or recovery and we use this capacity value to measure the impact of recovery on the cloud system.

The way of obtaining the response time \((W_i)\) during each cloud API is based on the model of the relationship between system workload and average response time of system requests. The “workload-response time” model is obtained by our empirical study and it is shown in Fig. 5 (blue curve). First we determine the system workload threshold (the maximum number of simultaneous requests allowed per second per VM) and then we acquire the average response time for all the possible workloads that are within the workload threshold. The workload threshold is determined by looking at to what extent of workload the system will be able to handle all the requests without abandoning any of them. Through our experiment, we find out the workload threshold to be 360 simultaneous requests per second per VM in the cloud. This can be explained by the fact that the buffer size of the Tomcat service which hosts TPC-W is limited (2048 bytes) and the queue size of Tomcat is by default 100. Actually, after load balancing the cloud system’s usual workload is within 50 simultaneous requests per second per VM if there is no error occurring and recovery. If errors occur and recovery is being performed, the system’s workload might go beyond 50 simultaneous requests per second per VM during the recovery depending on how many in-service VMs are terminated during certain steps of the recovery process. If the workload goes beyond 50 simultaneous requests per second per VM during recovery, we also need to know the average response time of these requests in order to calculate the recovery impact. Hence, in the model we allow the maximum workload to be 360 simultaneous requests per second per VM and this will benefit the calculation of recovery impact during recovery.

![Fig. 5. Workload-Response Time Model.](image)

The original model (Fig. 5, blue curve) can be represented as a linear regression model. Linear regression can not only reduce the noise of average response time for some workloads, but also provide us with a linear model with which we can calculate the response time based on the known workload as the input. The red line (Fig. 5) is the linear regression of the original workload-response time model. We obtain this model by using Matlab regression library[18]. It is represented by the below equation:

\[
w = 0.1497 \times r + 0.483225,\]

where \(w\) is the response time and \(r\) is the workload (number of simultaneous requests per second per VM).

To evaluate the performance of the linear regression, we compute the mean squared error and the squared correlation coefficient value. The former is 0.584893 and the latter is 0.99804. Now, we can use this linear model to compute the average response time for a certain workload that does not exceed the workload threshold.

The execution time \((T_i)\) of each cloud API is obtained by our empirical study[5]. We obtained the execution time for all the cloud APIs that could be used by recovery. For example, the execution time of the cloud API of “TerminateInstances” takes about 30 seconds.

To calculate the impact of a recovery action on the cloud system, we need to know what cloud APIs are included in the recovery action, and we also need to know the initial workload of the system as well as the initial number of VMs before the recovery starts to execute and these information are passed to the recovery service by the error detection service[4]. The algorithm of calculating the impact value of a recovery action is described as follows: Step 1) Use the inputs of initial workload and initial VM number to calculate the matrix \(W\) for the list of APIs composing the recovery action; Step 2) Determine the matrix \(T\) for the list of APIs in the recovery action; Step 3) Compute the impact value by multiplying the two matrices of \(W\) and \(T\), and return the result. We use an example to explain the calculation.

Suppose we have a recovery action consisted of two cloud APIs: 1) deregister two instances from ELB (API1) and 2) register two new instances with ELB (API2). Suppose the initial workload is 20 requests per second per VM and the initial VM number is 10. Hence, based on the “Workload-Response Time” model, the before recovery starts the system average response time is 3.5 seconds \((W_1 = 3.5\ s)\). After executing the API1, the workload becomes 25 \((20 \times 10 / (10 - 2)\) and the response time becomes 4.2 seconds \((W_2 = 4.2\ s)\). API1’s execution time is 3 seconds \((T_1 = 3\ s)\) and API2’s execution time is 3 seconds \((T_2 = 3\ s)\). The recovery impact is calculated to be \(23.1\ s^2 (W_1 \times T_1 + W_2 \times T_2)\).

D. Selecting Acceptable Recovery Action

Given a set of recovery actions, \(A = \{A_i\}\) and \(|A| = n\), operators must make a decision on which recovery action should be selected. From the above presentation, we know that each recovery action must be with three metrics, the time \(T_i\), the cost \(C_i\), and the impact \(I_i\), which are known beforehand. With respect to the three metrics, we should resort to Pareto optimality if there is no utility function that syntheses three objective into a single objective. Several techniques are available to find an exact Pareto optimum, such as weighted-sum method, \(\varepsilon\)-constraints method, and
Programming methods[19]. However, in this paper we do not luxuriously assume that users can always provide extra information like weights or constraints, and it is trivial to satisfy just the constraints by a simple search algorithm even if users can provide them. As we know from the literature, the time complexity of finding all strong Pareto optima, i.e. the Pareto set, is \( O(n^2) \) in general. The Pareto set is \( \{ A' | \langle A', T_i < T, C_i < C', I_i < I \rangle = \emptyset \} \). Thus, when \( n \) is not too big, it is affordable to find the Pareto set. After we have the Pareto set, whatever users’ additional constraints and requirements over the metrics are, we can always find one optimal solution from the Pareto set, the size of which is usually smaller than \( n \). In below, we show an \( O(n^2) \) algorithm to find the Pareto set. Specifically, the symbol of “\( \leq \text{TG} \)” in the algorithm means that the values of the metrics from an action are all smaller than or equal to those of the other action. There could be more than one action returned by the algorithm. The recovery framework can further select the action with least recovery time or least monetary cost or least recovery impact from the action list, depending on the business-specified importance of the metrics.

### VI. EXPERIMENT & EVALUATION

We evaluate our recovery service by using Asgard rolling upgrade[7] as the case study. We perform rolling upgrade operation on AWS EC2 cloud. Our recovery method is designed to cater for different cloud providers such as AWS and Windows Azure, etc. Since the cloud properties and functionalities required by our recovery method are shared among various cloud providers, we just use AWS as our target cloud platform to evaluate our recovery method. The evaluations on other cloud platforms just resemble AWS. First we describe the experimental environment and then we present the results of applicable recovery patterns filtering, recovery actions evaluation metrics calculation and recovery actions selection.

#### A. Experimental Environment

Fig. 6 shows the experimental environment. The cloud application we use is TPC-W which is a 3-tier web benchmark application running in Tomcat service. The web servers which run TPC-W are attached to an auto scaling group and are registered in an elastic load balancer. The average workload on each instance is 50 requests per second, and the total instances number for the web servers is 8. The Asgard service runs in a dedicated server and cloud operator’s own machine runs the client side of Asgard. Logs generated by Asgard and cloud will be collected by LogStash service[10] running in a LogStash server. Error detection and diagnosis service runs in the Error Detection & Diagnosis server and it relies on the logs collected to do error detection and diagnosis. Our recovery service runs in the Recovery server. It is triggered by the error detection service. The errors injected are from real-world experiences.

\[ \text{Algorithm 1: A search algorithm for the Pareto set} \]

\[
\begin{align*}
\text{Input:} & \quad \text{All actions } A_i \in A \\
\text{Output:} & \quad \text{The Pareto set } P
\end{align*}
\]

\[
\begin{align*}
1 & \quad P = P \cup A; \\
2 & \quad \text{repeat} \\
3 & \quad \quad \text{If } A_i \leq \text{TG} \text{ then} \\
4 & \quad \quad \quad \text{Replace } P_i \text{ with } A_i \text{ or delete } P_i \text{ after a replacement;} \\
5 & \quad \quad \text{end} \\
6 & \quad \quad \text{until all } P_i \in P 	ext{ have been visited;} \\
7 & \quad \quad \text{If } A_i \text{ has not replaced one in } P \text{ then} \\
8 & \quad \quad \quad P = P \cup A; \\
9 & \quad \quad \text{end} \\
10 & \quad \quad \text{until all } A_i \in A \text{ have been visited;} \\
11 & \quad \text{return } (P)
\end{align*}
\]

#### B. Applicable Recovery Patterns Filtering

The applicable recovery patterns for each step of Asgard rolling upgrade determined by our recovery service are shown in Table IV. Particularly, for step 6 (Wait for ASG to Start New Instance) we injected two types of errors: 1) VM launch fails and 2) VM launched but its image version is incorrect. For all the other steps we only injected one type of error. We show the applicable recovery patterns for each step determined by our recovery service. We also show the expected recovery patterns for each step determined by the business. For convenience, we use recovery pattern IDs to represent the eight recovery patterns, as shown in the legends of the table. We can see that the accuracy rate of the recovery patterns filtering for each step is 100%.

<table>
<thead>
<tr>
<th>Process Step</th>
<th>Errors</th>
<th>Applicable Recovery Patterns</th>
<th>Expected Recovery Patterns</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step 1. Create New Launch Configuration LC</td>
<td>LC creation fails</td>
<td>RP1, RP3, RP5, RP6</td>
<td>RP1, RP3, RP5, RP6</td>
<td>100%</td>
</tr>
<tr>
<td>Step 2. Update Auto Scaling Group</td>
<td>ASG not attached with LC but attached with LC&quot;</td>
<td>RP1, RP3, RP5, RP6, RP8</td>
<td>RP1, RP3, RP5, RP6, RP8</td>
<td>100%</td>
</tr>
</tbody>
</table>

**TABLE IV. APPLICABLE RECOVERY PATTERNS**

<table>
<thead>
<tr>
<th>Recovery Pattern ID</th>
<th>Recovery Pattern</th>
<th>Recovery Pattern ID</th>
<th>Recovery Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>RP1</td>
<td>Compensated Undo &amp; Redo</td>
<td>RP5</td>
<td>Reparation</td>
</tr>
<tr>
<td>RP2</td>
<td>Compensated Undo &amp; Alternative</td>
<td>RP6</td>
<td>Direct Redo</td>
</tr>
<tr>
<td>RP3</td>
<td>Rewind &amp; Replay</td>
<td>RP7</td>
<td>Direct Alternative</td>
</tr>
<tr>
<td>RP4</td>
<td>Rewind &amp; Alternative</td>
<td>RP8</td>
<td>Farther Undo &amp; Redo</td>
</tr>
</tbody>
</table>
C. Recovery Actions Evaluation & Selection

Recovery action selection is based on the Pareto set searching algorithm mentioned in section V. The selection results are shown in Table V. We assume recovery time is the most important metric. We report the selected recovery action for each step of rolling upgrade, and we show the calculated metrics values for the selected recovery action for each step. We also show the measured actual metrics values of each step’s selected recovery action when it is actually executed, and we compare them with the computed metrics values. For each cloud API in the recovery action, the real execution time fluctuates around its average execution time, hence there is a difference between the actual recovery time and the computed one. For example, the computed recovery time for step 5’s selected recovery action is 3 seconds versus real recovery time of 2.89 seconds. Since the actual recovery impact value during the recovery is also fluctuating around its average value, the actual recovery impact value is different from the computed one. For example, the calculated recovery impact of step 5’s selected recovery action is 27.12 s versus its measured actual value of 27.9 s. Such variance does not affect the optimal recovery action selection.

<table>
<thead>
<tr>
<th>Process Step</th>
<th>Errors</th>
<th>Applicable Recovery Patterns</th>
<th>Expected Recovery Patterns</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step 3. Sort Instances</td>
<td>Terminatior policy is different from user specification</td>
<td>RP1, RP3, RP5, RP6, RP8</td>
<td>RP1, RP3, RP5, RP6, RP8</td>
<td>100%</td>
</tr>
<tr>
<td>Step 4. Deregister Old Instance from ELB</td>
<td>Deregister VM from ELB fails</td>
<td>RP1, RP3, RP5, RP6, RP8</td>
<td>RP1, RP3, RP5, RP6, RP8</td>
<td>100%</td>
</tr>
<tr>
<td>Step 5. Terminate Old Instance VM</td>
<td>Terminate VM fails (too long termination time)</td>
<td>RP1, RP2, RP3, RP5, RP6, RP7, RP8</td>
<td>RP1, RP2, RP3, RP5, RP6, RP7, RP8</td>
<td>100%</td>
</tr>
<tr>
<td></td>
<td>2. VM’ launched but with wrong version image</td>
<td>RP1, RP2, RP3, RP4, RP5, RP6, RP7, RP8</td>
<td>RP1, RP2, RP3, RP4, RP5, RP6, RP7, RP8</td>
<td>100%</td>
</tr>
<tr>
<td>Step 7. Register New Instance with ELB</td>
<td>Register VM’ to ELB fails</td>
<td>RP1, RP3, RP5, RP6, RP8</td>
<td>RP1, RP3, RP5, RP6, RP8</td>
<td>100%</td>
</tr>
</tbody>
</table>

C. Recovery Actions Evaluation & Selection

We assume recovery time is the most important metric. We report the selected recovery action for each step of rolling upgrade, and we show the calculated metrics values for the selected recovery action for each step. We also show the measured actual metrics values of each step’s selected recovery action when it is actually executed, and we compare them with the computed metrics values. For each cloud API in the recovery action, the real execution time fluctuates around its average execution time, hence there is a difference between the actual recovery time and the computed one. For example, the computed recovery time for step 5’s selected recovery action is 3 seconds versus real recovery time of 2.89 seconds. Since the actual recovery impact value during the recovery is also fluctuating around its average value, the actual recovery impact value is different from the computed one. For example, the calculated recovery impact of step 5’s selected recovery action is 27.12 s versus its measured actual value of 27.9 s. Such variance does not affect the optimal recovery action selection.

<table>
<thead>
<tr>
<th>Step</th>
<th>Operation</th>
<th>Selected Recovery Action</th>
<th>Computed Value</th>
<th>Real Run Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step 2. Update Auto Scaling Group</td>
<td>Update ASG again with LC’; (Reparation)</td>
<td>3  0  23.91  2.99  0  22.47</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Step 3. Sort Instances</td>
<td>Update ASG with termination policy set by user; (Reparation)</td>
<td>3  0  23.91  3.17  0  25.55</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Step 4. Deregister Old Instance from ELB</td>
<td>Deregister old instance from ELB again; (Direct Redo)</td>
<td>3  0  23.91  3.24  0  24.78</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Step 5. Terminate Old Instance VM</td>
<td>Detach VM from ASG; (Direct Alternative)</td>
<td>3  0  27.12  2.89  0  27.9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Step 6. Wait for ASG to Start New Instance VM</td>
<td>Add hedged instance into ASG; (Direct Alternative)</td>
<td>3  0.0031  27.12  3.08  0.0031  28.87</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Step 7. Register New Instance with ELB</td>
<td>Register new instance with ELB again; (Direct Redo)</td>
<td>3  0  27.12  3.22  0  28.89</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

VII. THREATS TO VALIDITY

First, the recovery actions for each applicable recovery pattern are currently manually generated. We believe it is possible to automatically generate them. For example, context dependent specificity can be handled through approaches like domain specific language techniques coupled with a knowledge base with reusable skeleton code generation engine. We will address this in our future work.

Second, while the TPC-W benchmark is highly representative of a broad class of web based applications, it does not capture all modern cloud computing workload types. There is an emerging body of research work being conducted in the area of developing better cloud specific benchmarks, and we will aim to supplement the calculation of recovery impact with some of these newer and potentially more appropriate benchmarks.

VIII. RELATED WORK

A. Test Driven Scripts

OpsCode scripts such as Chef[20] can be used for implementing consumer-initiated cloud sporadic operations. To achieve high reliability, operation scripts can be written in a test driven manner, e.g. by using mini tests[21].
Specifically, mini tests test the functionality and availability of a module in amongst the whole script automated infrastructure. For example, a mini test for a module of shutting down Tomcat service can be conducted to check if Tomcat service will really be shut down successfully. Mini tests are carried out on a purposely built test bed. The errors and failures arising from operations running on the test bed could be recovered manually by the operator during the testing phase. However, the test bed is different from the actual operation runtime environment, so full guarantee cannot be made.

B. Cloud Operation Undoability

Consumer side cloud operators work with cloud systems with limited visibility and control. Some of these operations are undoable which means once an operation has executed, it is not reversible. Cloud undoability checker[22] has been proposed to help cloud operators to now manage cloud resources with a safety net. The facility to undo a collection of changes, reverting to a previous acceptable state, is widely recognized as valuable support for building dependable systems[22]. By using an abstract model of the effects of each available operation, we can check to which degree each operation is undoable. Undoability checker is able to identify which operations are not undoable and why. If undo is possible and desired, an AI planning technique[23] can be applied to automatically create a workflow that takes the system back to the desired earlier state.

C. Data Derivation Graph for System Recovery

Generating recovery actions for applicable recovery patterns relies on inputs including: the current system state, the previous consistent system state and the expected system state. The Data Derivation Graph (DDG)[24] is a proven method for achieving Undo that takes a similar approach. DDG records how data is produced by a running process and documents data flowing through steps agent execution details, resultant outputs[24]. DDG is automatically generated during process execution and is the main artefact driving process recovery[24].

IX. Conclusion & Future Work

Nowadays, sporadic operations (e.g. installation, upgrade, and reconfiguration) on cloud are being performed more frequently, due to the development of DevOps and requirements of continuous delivery. These sporadic operations are prone to failures due to several reasons such as cloud APIs uncertainty. Such failures are the major contributors to outages in cloud. In this paper, we proposed a methodology to address this problem. Based on the eight recovery patterns proposed by us, our recovery framework filters applicable recovery patterns, evaluates the recovery actions based on three evaluation metrics and selects the optimal recovery actions by using a Pareto set searching algorithm. We demonstrated our methodology through the case study of Asgard rolling upgrade on AWS EC2 cloud. Our future work includes: 1) work on automated generation of recovery actions; 2) provide exploration on more benchmarking tools.

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

[10] Logstash Official Website: http://logstash.net/ (last access time: 10th June 2015, 13:50).
[16] TPC-W Official Website: http://www.tpc.org/tpcw/ (last access time: 10th June 2015, 14:40).