Runtime Recovery Actions Selection for Sporadic Operations on Public Cloud

Min Fu$^{1,2}$, Liming Zhu$^{1,2}$, Daniel Sun$^{1,2}$, Anna Liu$^{1,2}$, Len Bass$^{1,2}$, Qinghua Lu$^{1,3}$

Software Systems Research Group, NICTA (Data 61, CSIRO), Sydney, Australia$^1$
School of Computer Science and Engineering, University of New South Wales, Sydney, Australia$^2$
College of Computer and Communication Engineering, China University of Petroleum, Qingdao, China$^3$


SUMMARY

Sporadic operations such as rolling upgrade or machine instance redeployment are prone to unpredictable failures in the public cloud largely due to the inherent high variability nature of public cloud. Previous dependability research has established several recovery methods for cloud failures. In this paper, we first propose eight recovery patterns for sporadic operations on public cloud. We then present the filtering process which filters applicable recovery patterns. We propose an automation mechanism to automatically generate recovery actions for those applicable recovery patterns based on our resource state transition algorithm. We also propose a methodology to evaluate the recovery actions generated for the applicable recovery patterns based on the recovery evaluation metrics of Recovery Time, Recovery Cost and Recovery Impact. This quantitative evaluation will lead to selection of the acceptable recovery actions. We propose two recovery actions selection mechanisms: one is based on user constraints of the recovery evaluation metrics, and the other one is based on Pareto set searching algorithm. We implement a recovery service and illustrate its applicability by recovering from errors occurring in the rolling upgrade operation on AWS cloud.

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

1. INTRODUCTION

Sporadic operations on public cloud refer to the deployment or maintenance operations which are relatively less frequent and regular than normal activities like transactions in an e-commerce application. Failures could happen in sporadic operations on public cloud due to the uncertainty of public cloud. Automatic recovery from failures during sporadic operations on public cloud such as rolling upgrade or machine instance (virtual machine) redeployment has become increasingly important because of the need to manage the uncertainty on public 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 applicable. For example, for the operation of deleting a data drive, Rewind & Replay or Compensated Undo & Redo are not applicable because the resource deletion operation cannot be rewound or undone. Similarly, the operation of “assigning IP address” is a process largely controlled by cloud service provider[6], hence the recovery pattern of Direct Redo is not applicable. Part of our solution here is that we have built up a knowledge base of these infeasible activities 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 automatically generate their recovery actions, 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. Each metric is meaningful to the business and 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 selecting the acceptable recovery action based on two selection methods: 1) using a Pareto set searching algorithm to search for the optimal recovery action based on the predicted values of the recovery evaluation metrics and 2) using user constraints based search method to select the acceptable recovery action based on the predicted values of the recovery evaluation metrics.

We constructed a prototype recovery service with the following capabilities: 1) it filters the applicable recovery patterns; 2) it automatically generates recovery actions for applicable recovery patterns based on our resource state transition algorithm; 3) it computes the values of recovery evaluation metrics for the generated recovery actions within applicable recovery patterns; and 4) it selects an acceptable recovery action for execution at runtime if failures occur. The recovery service is triggered by the error detection service named POD-Diagnosis[4]. 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 acceptable recovery actions that can 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, automation of recovery actions generation and the selection of an acceptable recovery action that can fulfil 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 two recovery actions selection methods to solve our recovery actions selection problem. In our previous work[19], we only manually generated recovery actions and only proposed one recovery action selection mechanism which is based on Pareto set search algorithm. In this paper, we figure out the way to automatically generate recovery actions and propose an additional recovery action selection method which is based on user constraints specified by cloud operators.

The rest of this paper is organized as follows: section 2 presents the background; section 3 describes the eight recovery patterns; section 4 describes recovery evaluation metrics; section 5 presents our prototype recovery service; section 6 describes the experiment and evaluation; section 7 discusses threats to validity; section 8 is related work; and section 9 provides the conclusion and our future work.

2. Background

2.1. Service-Oriented Architecture

A service-oriented architecture (SOA) is an architectural pattern in computer software design in which application components provide services to other components via a communications protocol, typically over a network. A service is a self-contained unit of functionality, such as retrieving an online bank statement. By that definition, a service is an operation that may be discretely invoked. Services can be combined to provide the functionality of a large software application[29]. SOA makes it easier for software components on computers connected over a network to cooperate. Every computer can run any number of services, and each service is built in a way that ensures that the service can exchange information with any other service in the network without human interaction and without the need to make changes to the underlying program itself. The design pattern of our recovery methodology also follows the idea of SOA. In the design of our recovery methodology, there are two components: a dedicated error detection service and an independent error recovery service. The error detection service triggers the error recovery service once errors are detected. Such a design pattern can ease the maintenance of each service, and also puts little restrictions on what implementation language should be used for each service.

2.2. Cloud Sporadic Operations

Example sporadic operations on cloud include installation, upgrade and reconfiguration, etc. 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 de facto 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 Figure 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 seven 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 by 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.
2.3. 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 may not be executed successfully 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. In our previous research work[9], we conducted empirical studies to figure out failure rates of some cloud API functions, as shown in Table I. Some cloud APIs have higher failure rates than other cloud APIs, such as “RunInstances” with a failure rate of 3.1% and “TerminateInstances” with a failure rate of 3.9%. Some cloud APIs have lower failure rates than others, such as “AttachVolume” with a failure rate of 0.3%.

<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>

2.4. Operation Errors Detection

In our previous work related to error detection and diagnosis for sporadic operations on AWS cloud[4], we 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. POD-Diagnosis is able to detect the errors of operational steps as soon as possible, based on analysing the runtime logs generated for each operational step in a real-time manner. Once an error is detected, it immediately triggers the recovery service. That is how runtime recovery is achieved. If POD-Diagnosis detects errors late (i.e. the errors are detected after a number of steps), then the recovery becomes very difficult or even impractical because the recovery service loses the track of the step that is currently being executed. Hence, in this case it will not trigger the recovery service. In other words, this paper only focuses on the cases where errors for each operational step are detected as soon as possible. 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

![Figure 1. Asgard Rolling Upgrade Operation.](image)
5) the current step specification and the specification of the last step prior to the current step. 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.

3. **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.

Figure 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 resource states are the inputs to the recovery: $S_{err}$, $S_{c1}$, $S_{c2}$, $C_i$, 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 $Step X$; $C_i$ 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 system state into the expected system state before the current step and then re-execute the current step ($S_{err}\rightarrow S_{r}\rightarrow Step X\rightarrow S_2$). 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 ($S_{err}\rightarrow S_{r}\rightarrow Alternative X\rightarrow S_2$). Rewind & Replay means to make the current erroneous system state into the consistent system checkpoint before the current step and then re-execute the current step ($S_{err}\rightarrow C_i\rightarrow Step X\rightarrow S_2$). Rewind & Alternative means to make the current erroneous system state into the consistent system checkpoint before the current step and then execute the alternative of the current step ($S_{err}\rightarrow C_i\rightarrow Alternative X\rightarrow S_2$). Reparation means to directly make the current erroneous system resource state into the expected system resource state for the current step ($S_{err}\rightarrow S_0$). Direct Redo means to directly re-execute the current step ($S_{err}\rightarrow Step X\rightarrow S_2$). Direct Alternative means to directly execute the alternative of the current step ($S_{err}\rightarrow Alternative X\rightarrow S_2$). Farther Undo & Redo means to make the current erroneous system resource state into the expected system resource 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 ($S_{err}\rightarrow S_{r}\rightarrow Step X-1\rightarrow Step X\rightarrow S_2$). 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 resource state ($S_{us}$). 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.
The eight recovery patterns are based on all the possible paths and associated mechanisms from an erroneous resource state \((S_{\text{err}})\) during the operation to the expected resource state \((S_{\text{e}})\). To increase the efficiency of state management, we introduced the concept of operation resource space, which means the set of manipulated resources during all the steps of an operation on public cloud. The inputs needed for determining resource space are cloud API tracking logs which are generated automatically within all kinds of public cloud. Cloud API tracking logs record all the cloud APIs that have been called for a sporadic operation as well as the parameters of each API call. The cloud API tracking logs across different public clouds follow the same structure which consists of the timestamps of all relevant API calls, the signatures of all relevant API calls and the parameters of all relevant API calls. For AWS cloud, the key inputs needed for determining resource space are CloudTrail logs\[6\] which contain all the cloud APIs having been called in the past operation runs. Figure 3 shows an example of CloudTrail logs. It records the track of the cloud API of “UpdateAutoScalingGroup” in JSON format. It specifies the API call time (“eventTime”), API function name (“eventName”), API call parameters (“requestParameters”) and API call response information (“responseElements”).

```json
```

Figure 3. CloudTrail Log.

Generating the resource space relies on the information provided by cloud API tracking logs. Since cloud API tracking logs provided by different public cloud platforms semantically have the same structure, the mechanism of resolving cloud API tracking logs in one cloud is just the same as that of resolving cloud API tracking logs in another different cloud. Taking AWS as the example, the resource space is generated by using AWS CloudTrail logs. The approach for generating resource space on AWS is illustrated in Figure 4. First, based on the operation process with timestamps and CloudTrail logs, all of the operation related cloud API calls can be determined by correlating the timestamps in cloud trail logs with the start time and end time of the operational process. Second, based on the API-resource mapping table which provides resource changes for each cloud API (e.g. the resource change for API “CreateLaunchConfiguration” is “Launch Configuration”, and the resource change for API “TerminateInstances” is “Instances”), we can obtain the overall resource changes for the operation. But the resource changes acquired here may contain some repetitive items (e.g. two items of “instances”), and may not be represented in the correct resources dependency order (e.g. “instances” should be aligned under “ASG” because an ASG contains several instances). Such resources dependency relationship is specified in AWS documentation[6]. Hence, the next step is removing any repetitive resource items and reshuffling the items according to cloud resources dependency relationship specified by AWS documentation. Finally we save the operation resource space.

Figure 4. Operation Resource Space Determination.
Using rolling upgrade as an example, the resource space determined by the resource space determination method is illustrated in Figure 5. It contains four resource items: Instance, Launch Configuration (LC), Auto Scaling Group (ASG), and Elastic Load Balancer (ELB). Each resource has its own attributes (e.g. instances have the attributes of instance id, instance type, instance image, etc.). The dependency relationships among those resources are: 1) one LC attaches with one ASG; 2) one ASG contains many instances; 3) one ELB contains many instances and 4) one instance can belong to several ELBs. These relationships also align with the resources dependency relationship specified by cloud resources relationship documentations such as the AWS documentation[6]. The cloud resources relationship documentations provided by different public clouds also have the same specification on the cloud resources dependency relationship. Hence, no matter on which public cloud the cloud operators perform the sporadic operation, the operation resource space determined by our method will be the same.

Expected resource state templates are generated using the past operation that is the same as the operation going to be performed, because the workflow and the steps of the new operation are exactly the same as the past operation. Figure 6 explains the method of generating expected resource state templates on AWS. First, from the process model (with timestamps) obtained by the process mining tool[4], we obtain the start time and end time of each step in the process of the past operation. Next, we obtain the cloud API calls and the API calls parameters for each step by analyzing CloudTrail logs information and operational steps timestamps information. For AWS, specifically, we correlate the timestamps of APIs in cloud trail information with the timestamps of operational steps to find out what API calls are involved in each step. Then we capture the initial resource state of the cloud system before the operation starts by using the resource state capturing service. Next, based on API calls information for each step and the initial resource state before the operation starts, the expected resource state template after each step is generated. Specifically, the expected state after step 1 is the returned result of applying step 1 related APIs onto the initial state (e.g. LC is created), the expected state after step 2 is the returned value of applying step 2 related APIs onto the state after step 1(e.g. ASG is updated), and so on and so forth.
Figure 7 shows the example of the generated expected resource state template after step 1 (Create New LC) of rolling upgrade. Before the operation starts, the initial state of the resources is captured in XML format. Then after applying the cloud API function of “CreateNewLaunchConfiguration”, one resource is changed. The changed resource is the “new LC” (in underlined bold font), and hence it evolves the initial resource state by adding the “new LC”. Then we apply the similar logic to other operational steps as well. In this paper, we do not show the generated resource state templates for other steps. Each generated resource state template follows the tree structure of the determined resource space. In the generated resource state templates, some values are missing (e.g. id of the new launch configuration is unknown yet), and they are represented as question marks (“?”). In the operation’s runtime, the missing values will be populated and replaced by the real values obtained from runtime operation logs. We save the generated expected resource state templates into the xml file which can be retrievable by the recovery service during recovery.

<table>
<thead>
<tr>
<th>Operation Phase</th>
<th>Expected System State Template Generated</th>
</tr>
</thead>
</table>
| Before Operation Starts | `<state id="initialState">
  <Instance>
    <instance id="i-b8ca0eb0" state="running" ami="ami-9fe577af" type="t1.micro"/>
    <instance id="i-b9ca0eb1" state="running" ami="ami-9fe577af" type="t1.micro"/>
    <instance id="i-baca0eb2" state="running" ami="ami-9fe577af" type="t1.micro"/>
  </Instances>
  <LCs>
    <LC id="windowsgroup-20130803194139" ami="ami-9fe577af" instancestype="t1.micro"/>
  </LCs>
</state>` |
| After Execution of Step 1 (Create New LC) | `<state id="stateForStep1">
  <Instance>
    <instance id="i-b8ca0eb0" state="running" ami="ami-9fe577af" type="t1.micro"/>
    <instance id="i-b9ca0eb1" state="running" ami="ami-9fe577af" type="t1.micro"/>
    <instance id="i-baca0eb2" state="running" ami="ami-9fe577af" type="t1.micro"/>
  </Instances>
  <LCs>
    <LC id="windowsgroup-20130803194139" ami="ami-9fe577af" instancestype="t1.micro"/>
  </LCs>
</state>` |

The states of the resources during an operation are captured by the resource state capturing service. Unlike existing traditional checkpointing mechanisms which capture the states for the whole software system[25], our state capturing method only captures the states of the resources involved in the determined resource space of the operation rather than the resources of the whole cloud system, hence increasing the efficiency of state capturing especially when the system scale is large. The state capturing algorithm is shown in Figure 8. It captures the state information of each resource item that is included in the resource space by describing the information of each cloud resource. The resource state capturing method captures the states of cloud system resources in sequence. The efficiency of describing resources information sequentially is acceptable. To achieve higher efficiency of state capturing, a better way is to capture the states of multiple resources within the determined resource space in parallel. For example, describing instances information by issuing “DescribeInstances” cloud API can be executed simultaneously with describing elastic load balancers (ELB) information by issuing “DescribeLoadBalancers” cloud API. After each recovery point and after the recovery for each recovery point, the state capturing service will be triggered to capture the resource state, stored in the XML format.
4. 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 an E-commerce website due to error recovery may result in loss of tens of thousands of dollars. So Recovery Time (RT) is the first one of the metrics that we use to evaluate recovery actions. Businesses often operate with a specific Recovery Time Objective (RTO)[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 the recovery for sporadic operations on the cloud systems. In the cloud context, recovery actions such as launching a new instance will incur additional monetary cost[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 systems[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 bad consequence on the performance of each cloud application[13]. Hence, reducing the recovery’s negative performance impact on the cloud systems 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 systems, recovery time is defined as the time for a system to recover from a failure to an agreed service level[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 the sporadic operation) to return from a failure to the potentially degraded SLA/capacity; and 2) the time for the operation to return from a failure (or erroneous state) to an acceptable consistent 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 each consist of a set of relevant cloud APIs.

2) Recovery Cost (RC). It means the money charged by AWS during recovery for the cloud APIs (e.g. RunInstances) which composite the recovery action. We can get this information from the pricing policies in AWS website[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 this 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 to 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.

5. OUR PROTOTYPE RECOVERY SERVICE

Our recovery service is designed with the aim of satisfying the recovery requirements, which are Recovery Time Objective (RTO), Recovery Cost Objective (RCO) and Recovery Impact Objective (RIO). According to our definitions, RTO means that the recovery time should not exceed a specified time boundary; RCO means the monetary cost of recovery should not exceed a specified monetary value; RIO means the recovery’s negative impact on the cloud system should not exceed a specified impact value. The overview of our recovery service prototype is described in Figure 9. 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 filter the applicable recovery patterns by performing state reachability checking, idempotence checking and step alternative existence checking. Each applicable recovery pattern could contain one recovery action. Hence, after filtering the applicable recovery patterns we automatically generate the recovery action for each applicable recovery pattern. Certain recovery actions within applicable recovery patterns might not be acceptable because they may fail in satisfying the recovery requirements. We compute the values of the recovery evaluation metrics for each recovery action. The recovery evaluation metrics are Recovery Time, Recovery Cost and Recovery Impact. Finally, we select the acceptable recovery action which can satisfy the recovery requirements 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 similar cloud APIs provided, have instances (virtual machines) running on them, and incur monetary costs.

![Figure 9. Overview of Recovery Service.](image-url)

5.1. Applicable Recovery Patterns Filtering

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 no alternative step exists, 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 filter 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 activities involved in the state transition in the context of cloud are not feasible. In the context of sporadic operations on cloud, the state transition involving the activities of stateful data drive (or store) creation and IP address reassignment is 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 the relevant cloud API function. Reassigning a new IP address to a cloud instance is not feasible because this activity by cloud consumers is not allowed by public cloud platforms due to the limited visibility and indirect 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 resource state structure consists 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 the current erroneous state is, 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 the rolling upgrade operation; no matter what the current state is after this step (e.g. auto scaling group is attached with another unknown launch configuration), 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 the 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 filtering 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 ($S_1$) before the current step; 3) the captured consistent state ($C_1$) before the current step; 4) the expected state ($S_2$) 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 ($S_0$) 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 $S_1$ 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 $S_2$ is reachable form $S_{err}$ and if alternative of $Step X$ exists, and if so Compensated Undo & Alternative will be included in the applicable recovery patterns list. For Rewind & Redo, we check if $C_1$ is equal to $S_1$ (because $C_1$ might be different from $S_1$ due to state capturing service delay or false positives of state capturing; if so $C_1$ is not the current state and Rewind is invalid) and if $C_1$ is reachable from $S_{err}$, and if so Rewind & Redo will be included in the applicable recovery patterns list. For Rewind & Alternative, we check if $C_1$ is equal to $S_1$, if $C_1$ 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 $S_2$ 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 $S_0$ 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.

5.2. 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 automatically generated. The automation mechanism just needs to determine what actions need to be performed in order to transit current erroneous system resource state into the goal state (resource state transition), and what operational steps or alternatives need to be executed. For those recovery actions where no resource state transition is needed (e.g. Recovery Pattern of Direct Redo), the automation mechanism just needs to execute the operational steps or alternatives. Taking the recovery pattern of Rewind & Replay as an example, it first makes the system go from the current erroneous resource state into the previous consistent resource state, followed by re-executing the current step. The automation mechanism for resource state transition is called Resource State Transition Algorithm. This algorithm is described in Figure 10. It first compares the root node in the current state and the root node in the goal state. If there is difference (e.g. Launch Configuration is in goal state but not in current state), relevant cloud APIs will be called in the function of “TransitStateItem” to fix the difference (e.g. API of “CreateLaunchConfiguration”). If both the root node of current state and the root node of goal state have child nodes, it obtains the first child node of current state and the first child node of the goal state. These two child nodes follow the same structure as their parent nodes and hence the algorithm makes transition between these two child nodes by recursively calling the state transition function itself. The same logic happens to the second child node of current state and the second child node of goal state, to the third node of current state and the third node of goal state, and so on and so forth, until both the current state and goal state have no child nodes. 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 much time. The reason for this error to
Although the same cloud instance termination API fails due to uncertainty and instability of cloud platform. This is a transient API-related error on cloud, so at the next time when the same API is called, it may not necessarily fail. Since the recovery patterns is Rewind & Replay, it first reverts the system back to the previous consistent state by terminating the old instance and launching another old instance, and then it re-executes the same step (Step 5) by terminating the newly launched old instance. The recovery action consists of several activities which are each mapped with relevant cloud APIs. For example, the activity of “Terminate old version instance” is mapped with the cloud API of “TerminateInstancesInAutoScalingGroup”. Although the same cloud API is called again, it won’t necessarily fail because it is totally independent of the previous call.

5.3. 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 consist 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 “Run Instances” 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 Figure 11. 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.

Figure 11(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 $S (S = S_1 + S_2 + S_3 + S_4 + S_5)$ represents the impact of the successful operation on the cloud system.

Figure 11(b) shows the response time trajectory of an erroneous operation where the error occurs in step 2 and the recovery action is taken for step 2. The error in step 2 makes the response time of step 2 another value (area to be $S_2'$), and the recovery action ($Recovery 2'$) introduces additional response time trajectory (area to be $SR_2$) over the original one. Hence, the overall
impact of this erroneous operation with recovery is \( S' \) which is equal to \( S_1 + S_2 + SR_2 + S_3 + S_4 + S_5 \). The impact of the recovery action is calculated to be \( S'' \), which is equal to \( (S_2' - S_2) + SR_2 \).

Figure 11(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 \( S_1 + S_2 + SR_2 + S_3 + S_4 + S_5 \), and the impact of the recovery action in this case is calculated to be \( S'' \), which equals to \( (S_2' - S_2) + SR_2 \).

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 \( (1 - Pr(fp)) \times (S_2' - S_2) + Pr(fp) \times SR_2 \), which is equal to \( (1 - Pr(fp)) \times (S_2' - S_2) + (1 - Pr(fp)) \times SR_2 + Pr(fp) \times SR_2 \). 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 \( 1 \leq i \leq n \) 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 \( 1 \leq i \leq n \). This formula can be further represented as below:

\[
I (\text{Impact}) = W \times T,
\]

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.

System response time \( (W) \) 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 \( W \) 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 is shown in Figure 12 (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 for the extent of workload to which 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 is reasonable given 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. Theoretically, if the Tomcat server could process each simultaneous request without any delay even if there are a large number of simultaneous requests, the workload threshold should be infinite. However, it is impossible that the Tomcat server in an instance can process large number of simultaneous requests without any delay. Hence, the queue in Tomcat is necessary and when there are a large number of simultaneous requests, some requests stay in the queue provided by Tomcat while other requests are processed by Tomcat service. That’s also why the maximum number of allowed simultaneous requests is usually larger than the queue size (100). According to our experimental settings, we found that the allowed maximum number of simultaneous requests on each instance is 360. 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.

![Figure 12. Workload-Response Time Model.](image)

The original model (Figure 12, 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 (Figure 12) 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:

\[
W = 0.1497 \times r + 0.483225, \quad \text{where } W \text{ is the response time and } r \text{ is the workload (number of simultaneous requests per second per VM).}
\]

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 \( T \) 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 this information is 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 matrixes of W and T, and return the result. We use an example to explain the calculation. Suppose we have a recovery action consisting 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 \((W_1 = 3.5 \text{ s})\). After executing the API1, the workload becomes 25 \((20 \times 10 / (10 - 2))\) and the response time becomes 4.2 seconds \((W_2 = 4.2 \text{ s})\). API1’s execution time is 3 seconds \((T_1 = 3 \text{ s})\) and API2’s execution time is 3 seconds \((T_2 = 3 \text{ s})\). The recovery impact is calculated to be \(23.1 \text{ s}^2 (W_1 \times T_1 + W_2 \times T_2)\).

5.4. Selecting Acceptable Recovery Action

We propose two selection methods for selecting the acceptable recovery actions: 1) Pareto set search based selection and 2) User constraints based selection. In our previous work[19], we only proposed one selection mechanism which is based on Pareto set search algorithm. In this paper, we proposed an additional recovery action selection method which is based on user constraints specified by cloud operators.

5.4.1. Pareto Set Search Based Recovery Action Selection

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 associated 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 synthesizes three objective into a single objective. Several techniques are available to find an exact Pareto optimum, such as weighted-sum method, \(\epsilon\)-constraints method, and programming methods[20]. However, 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 \(\{A_i | \forall A_j | T_i < T_j \text{ and } C_i < C_j \} \cup \{A_i | \forall A_j | I_i < I_j \} = \emptyset\). 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 \(n\). Below, we show an \(O(n^2)\) algorithm to find the Pareto set. Specifically, the symbol of \("\leq_{TCI}\)" 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.

<table>
<thead>
<tr>
<th>Algorithm 1: A search algorithm for the Pareto set</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> All actions (A_i \in A)</td>
</tr>
<tr>
<td><strong>Output:</strong> The Pareto set (P)</td>
</tr>
<tr>
<td>(P \leftarrow P \cup A_i;)</td>
</tr>
<tr>
<td>repeat</td>
</tr>
</tbody>
</table>
| \(\text{if } A_i \leq_{TCI} P_j \text{ then} \)
| \(\quad \text{Replace } P_j \text{ with } A_i \text{ or delete } P_j \text{ after a replacement};\) |
| \(\text{end} \)
| until all \(P_j \in P\) have been visited; |
| \(\text{if } A_i \text{ has not replaced one in } P \text{ set then} \)
| \(\quad P \leftarrow P \cup A_i;\) |
| \(\text{end} \)
| until all \(A_i \in A\) have been visited; |
| return \(P\) |

5.4.2. User Constraints Based Recovery Action Selection

Having obtained a list of applicable recovery actions with their evaluation metrics calculated, we need to select an acceptable recovery action from them. First, we need a selection rule or objective which is provided by cloud operators. The selection rule could be stated in a format that the value of each of the three metrics (Recovery Time, Recovery Cost and Recovery Impact) should be within a predefined objective or constraint. This rule also forms the recovery requirements for the business. We denote the constraint on Recovery Time to be \(Cons_{rt}\), the constraint on Recovery Cost to be \(Cons_{rc}\), and the constraint on Recovery Impact to be \(Cons_{ri}\). Hence, the acceptable recovery actions will be those whose recovery time is within \(Cons_{rt}\) and whose recovery cost is within \(Cons_{rc}\) and whose recovery impact is within \(Cons_{ri}\). It is possible that there is no acceptable recovery action found due to the fact that no recovery action satisfies all the three constraints. If this happens, we will try to find out those recovery actions which can satisfy two constraints simultaneously (\(Cons_{rt}\) and \(Cons_{rc}\) or \(Cons_{rt}\) and \(Cons_{ri}\) or \(Cons_{rc}\) and \(Cons_{ri}\)). If there is no recovery action found, we will try to find out those recovery actions which can satisfy one constraint (\(Cons_{rt}\) or \(Cons_{rc}\) or \(Cons_{ri}\)). Finally the first one in the acceptable recovery actions list is selected. This logic can be modelled in constraint programming as below:
Recovery Action Selection

Suppose there are \( n \) recovery actions for a certain step of operation and each recovery action is denoted by \( R_i \), where \( 1 \leq i \leq n \). The recovery actions list can be denoted by \( R \), where \( R = \{R_i | 1 \leq i \leq n\} \).

For each recovery action \( R_i \), the three metrics are denoted by \( RT_i \), \( RC_i \), and \( RI_i \).

The goal is to find the list of acceptable recovery actions denoted by \( R' \), where \( R' = \{R_i | 1 \leq i \leq n \& RT_i \leq Cons_t, \& RC_i \leq Cons_s, \& RI_i \leq Cons_a\} \).

If \( R' \) is empty, we try to find the list of acceptable recovery actions denoted by \( R'' \), where \( R'' = \{R_i | 1 \leq i \leq n \& RT_i \leq Cons_t, \& RC_i \leq Cons_s\} \).

If \( R'' \) is empty, we try to find the list of acceptable recovery actions denoted by \( R''' \), where \( R''' = \{R_i | 1 \leq i \leq n \& RC_i \leq Cons_s\} \).

If \( R''' \) is empty, we try to find the list of acceptable recovery actions denoted by \( R'''' \), where \( R'''' = \{R_i | 1 \leq i \leq n \& RI_i \leq Cons_a\} \).

If \( R'''' \) is empty, we try to find the list of acceptable recovery actions denoted by \( R'''', \) where \( R'''' = \{R_i | 1 \leq i \leq n \& RC_i \leq Cons_s\} \).

If \( R'''' \) is empty, we try to find the list of acceptable recovery actions denoted by \( R''''' \), where \( R''''' = \{R_i | 1 \leq i \leq n \& RI_i \leq Cons_s\} \).

If \( R''''' \) is empty, we try to find the list of acceptable recovery actions denoted by \( R'''''' \), where \( R'''''' = \{R_i | 1 \leq i \leq n \& RI_i \leq Cons_a\} \).

If \( R'''''' \) is still empty, then we assign \( R' \) with the value of \( R \).

Finally, \( R '\) (\( R' \in R'' \)) which is the first element of \( R'' \) is selected.

6. EXPERIMENT & EVALUATION

We evaluate our recovery service by using Asgard rolling upgrade[7] as the case study. We use rolling upgrade as our experimental cloud operation because it is the most frequently performed cloud operation due to continuous deployment practice in modern IT industries[1]. We perform the rolling upgrade operation on AWS EC2 cloud, which is one of the most popular public clouds in the world. Our recovery method is designed to cater for different public 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.

6.1. Experimental Environment

Figure 13 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 service runs in the Error Detection server. Our recovery service runs in the Recovery server.

![Experimental Environment](image)

Figure 13. Experimental Environment.
The recovery service is triggered by the error detection service once errors are detected. The errors injected are from real-world experiences. In Netflix company where rolling upgrade operations happen frequently every day, the cloud operators encounter outage of cloud APIs (such as failing to create LC) relatively more often[7]. For instance, the API of “TerminateInstancesInAutoScalingGroup” once failed 6 times within a single day when cloud operators from Netflix performed more than 30 rolling upgrade operations on the company’s movie website.

6.2. 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 IV. Applicable Recovery Patterns

<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''</td>
<td>RP1, RP3, RP5, RP6, RP8</td>
<td>RP1, RP3, RP5, RP6, RP8</td>
<td>100%</td>
</tr>
<tr>
<td>Step 3. Sort Instances</td>
<td>Termination 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, RP4, RP5, RP6, RP7, RP8</td>
<td>RP1, RP2, RP3, RP4, 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>
</tbody>
</table>

### Legends

<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>Further Undo &amp; Redo</td>
</tr>
</tbody>
</table>
6.3. Recovery Actions Evaluation & Selection

Recovery action selection is based on the two recovery selection methods mentioned in section 5. The selection results for the Pareto set search based selection method 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>Operation Step</th>
<th>Selected Recovery Action</th>
<th>Computed Value</th>
<th>Real Run Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>$RT$ (s)</td>
<td>$RC$ (s)</td>
</tr>
<tr>
<td>Step 1. Create New Launch Configuration LC’</td>
<td>Recreate new launch configuration LC’; (Direct Redo)</td>
<td>3 0</td>
<td>23.91</td>
</tr>
<tr>
<td>Step 2. Update Auto Scaling Group</td>
<td>Update ASG again with LC’; (Reparation)</td>
<td>3 0</td>
<td>23.91</td>
</tr>
<tr>
<td>Step 3. Sort Instances</td>
<td>Update ASG with termination policy set by user; (Reparation)</td>
<td>3 0</td>
<td>23.91</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</td>
<td>23.91</td>
</tr>
<tr>
<td>Step 5. Terminate Old Instance VM</td>
<td>Detach VM from ASG; (Direct Alternative)</td>
<td>3 0</td>
<td>27.12</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</td>
<td>27.12</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</td>
<td>27.12</td>
</tr>
</tbody>
</table>
The selection results for the user constraints based selection method are shown in Table VI. Recovery action selection is based on the recovery requirements specified by business stakeholders. Considering the features of Asgard rolling upgrade, we determine the recovery requirements for each step of the operation to be: 1) recovery time should not exceed 60 seconds; 2) recovery cost should not exceed 1 dollar; 3) recovery impact should be within 200 square seconds. We report the selected recovery action for each step of rolling upgrade, and we also 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 steps 1, 4 and 7, the selected recovery action belongs to the recovery pattern of Direct Redo; for steps 2 and 3, the selected recovery action belongs to the recovery pattern of Reparation; for step 5, the selected recovery action belongs to the recovery pattern of Direct Alternative; for step 6 with the first error, the selected recovery action belongs to the recovery pattern of Direct Alternative; for step 6 with the second error, the selected recovery action belongs to the recovery pattern of Rewind & Alternative. We can also observe that the selection results of the two selection methods are exactly the same, for the rolling upgrade case in our experiment. But it is not always the case, the selection results of these two recovery action selection methods and the difference of selection results are dependent on what is the operation performed on cloud. The real run values are different from the previous ones, because of cloud performance variance. Again, such variance won’t affect selection of the acceptable recovery actions.

**TABLE VI. RECOVERY ACTIONS EVALUATION & SELECTION RESULTS-USER CONSTRAINTS BASED**

<table>
<thead>
<tr>
<th>Operation Step</th>
<th>Selected Recovery Action</th>
<th>Computed Value</th>
<th>Real Run Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>RT (s)</td>
<td>RC (s)</td>
</tr>
<tr>
<td>Step 1. Create New Launch Configuration LC’</td>
<td>Recreate new launch configuration LC’; (Direct Redo)</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Step 2. Update Auto Scaling Group</td>
<td>Update ASG again with LC’; (Reparation)</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Step 3. Sort Instances</td>
<td>Update ASG with termination policy set by user; (Reparation)</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Step 4. Deregister Old Instance from ELB</td>
<td>Deregister old instance from ELB again; (Direct Redo)</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Step 5. Terminate Old Instance VM</td>
<td>Detach VM from ASG; (Direct Alternative)</td>
<td>3</td>
<td>0</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</td>
<td>0.0031</td>
</tr>
<tr>
<td>Step 7. Register New Instance with ELB</td>
<td>Register new instance with ELB again; (Direct Redo)</td>
<td>9</td>
<td>0.0031</td>
</tr>
</tbody>
</table>
7. Threats to Validity

First, 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.

Second, our work in this paper only focuses on public clouds such as AWS cloud. When applied to private clouds, our method may not work straightforwardly due to even more limited accessibility and control provided by private clouds. Nonetheless, recovery for private clouds is out of the scope of this paper, and we will address it in our future work.

8. Related Work

8.1. Test Driven Scripts

OpsCode scripts such as Chef[21] 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[22]. 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.

8.2. Cloud Operation Undoability

Consumer side cloud operators work with cloud systems with limited visibility and control. Some of these operations are not undoable which means once an operation has been executed, it is not reversible. Cloud undoability checker[23] 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[23]. 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[24] can be applied to automatically create a workflow that takes the system back to the desired earlier state, which is similar to rolling back a system state to a previous consistent system state in the context of message-passing systems[25].

8.3. 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)[26] 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[26]. DDG is automatically generated during process execution and is the main artefact driving process recovery[26].

8.4. Software Metrics Selection for Defect Prediction

In order to predict software defects, a verified strategy is to build software quality prediction models that estimate the quality of program models[27]. In order to build software quality prediction models in a more accurate and efficient manner, we need to select a useful set of software metrics. There are some existing researches on the selection of software metrics for building software quality prediction models, such as the Hybrid Attribute Selection Approach[27]. Exactly speaking, this approach is a pre-processing step for building the software defect prediction models, which also belong to software quality prediction models. Hybrid Attribute Selection Approach is a hybrid attribute selection approach consisting of features ranking and features subset selection[27]. It solves the problem that an exhaustive search for such metrics is usually not feasible due to limited project resources, especially if the number of available metrics is large[27]. In the context of evaluation for recovery actions, a large space of metrics can be explored. However, if we want to make a more efficient evaluation for recovery actions, metric selection is needed. Hence, Hybrid Attribute Selection Approach can also be applied on selecting a subset of metrics that should be used for evaluating a recovery action. We tried to apply this approach to search for a metric subset for recovery evaluation, but one problem is that it would rely on a large amount of historical recovery data with values of a variety of different metrics, and those data might contain much invalid information. Analysing these data based on machine learning techniques such as information gain (IG) or Support Vector Machine (SVM) is sometimes not accurate, hence yielding wrong outputs. Compared to Hybrid Attribute Selection Approach, the approach we use in this paper to propose the three recovery evaluation metrics is based on a combination of qualitative and quantitative analyses from the perspective of end users. Since our recovery actions are performed on sporadic operations which are initiated by end users, metrics that are related to end user expectations are more useful and insightful. Hence, it is better to avoid using statistically empirical methods such as Hybrid Attribute Selection Approach to determine the metrics subset for recovery evaluation.
8.5. Recovery in Business Process Execution Language (BPEL)

Business Process Execution Language (BPEL)[30] is a protocol for specifying the logic of business processes, e.g. shipping goods to overseas from a local store. It requires the BPEL engine to execute BPEL scripts, and error recovery mechanisms are specified as recovery blocks in the BPEL scripts[28][30]. The recovery block is usually specified in the form of a Compensation node, where the recovery flow in case of an error is provided[28][30]. Recovery methods like BPEL recovery mechanisms[28][30] usually deal with normal activities of cloud systems. An example of normal activities can be the transaction workflow of an E-Commerce website. One limitation is that it needs BPEL engine to run, which is impractical for sporadic operations on cloud. Moreover, BPEL recovery has limited recovery patterns[28][30] (e.g. compensation) and it is hard to recover for various failure scenarios within sporadic operations on cloud. Compared to BPEL recovery, our recovery service is focused on sporadic operations of cloud systems and it is intended for runtime recovery from different types of operational failures.

9. Conclusion & Future Work

Nowadays, sporadic operations (e.g. installation, upgrade, and reconfiguration) on public 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 API uncertainty. Such failures are the major contributors to outages in cloud. In this paper, we proposed a recovery methodology to address this problem. Based on the eight recovery patterns proposed by us, our recovery framework filters applicable recovery patterns, generates recovery actions for those applicable recovery patterns in an automated way, evaluates the recovery actions based on three recovery evaluation metrics and selects the acceptable recovery actions by using two recovery action selection methods: Pareto set searching algorithm and user constraints based selection mechanism. We demonstrated the feasibility of our recovery methodology through the case study of Asgard rolling upgrade on AWS EC2 cloud. Our future work includes: 1) extend our methodology to cater for private clouds; 2) provide exploration on more cloud benchmarking tools.

Acknowledgment

NICTA (Data 61, CSIRO) 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: 5th June 2016, 13:50).
[16] TPC-W Official Website: http://www.tpc.org/tpcw/ (last access time: 5th June 2016, 14:40).


