Process-Oriented Non-Intrusive Recovery for Sporadic Operations on Cloud

Min Fu, Liming Zhu, Ingo Weber, Len Bass, Anna Liu, Xiwei Xu
NICTA and Data61, CSIRO, Sydney, Australia
School of Computer Science and Engineering, University of New South Wales, Sydney, Australia

Abstract — Cloud-based systems get changed more frequently than traditional systems. These frequent changes involve sporadic operations such as installation and upgrade. Sporadic operations may fail due to the uncertainty of cloud platforms. Each sporadic operation manipulates a number of cloud resources. The accessibility of resources manipulated makes it possible to build an accurate process model of the correct behavior for an operation and its desired effects. This paper proposes a non-intrusive recovery approach for sporadic operations on cloud, called POD-Recovery. POD-Recovery utilizes the above-mentioned process model of the operation. When needed, it triggers recovery actions based on the model through non-intrusive means, i.e., without modifying the code which implements the sporadic operation. POD-Recovery employs an efficient artificial intelligence (AI) planning technique for generating recovery plans. We implement POD-Recovery and evaluate it by recovering from faults injected into 920 runs of five representative sporadic operations.

Keywords — Software Reliability; Cloud; Non-Intrusive Recovery; Log Analysis; DevOps;

I. INTRODUCTION

Sporadic operations, such as installation, upgrade, or reconfiguration, are a class of system operations which are performed irregularly. In contrast, normal activities, such as transactions in an e-commerce application, or monitoring, take place continuously or very frequently. However, due to movements like DevOps and continuous deployment, sporadic operations are becoming much more frequent than in the past: while traditionally upgrades of applications in production happened once per quarter or year, with DevOps 25 full deployments per day are the norm in some organizations[1]. DevOps, an abbreviated combination of Development and Operations, “is a set of practices intended to reduce the time between committing a change to a system and the change being placed into normal production, while ensuring high quality”[1]. According a report by IBM, DevOps has increased developer effectiveness from 58% to 80%[2]. Meanwhile, Gartner says that “by 2016, DevOps will evolve from a niche to a mainstream strategy employed by 25 percent of Global 2000 Organizations”[3].

The DevOps goals can best be achieved by using cloud computing, including practices like Infrastructure-as-Code, i.e., by implementing sporadic operations to manage virtual infrastructure through tools or scripts that invoke cloud APIs[1]. For example, the rolling upgrade operation can be implemented using cloud infrastructure APIs to upgrade an application without impacting its availability, by iteratively upgrading a small number of its virtual machines (VMs) at a time[4]. However, such operations can fail — and executing them frequently means that even unlikely errors have to be considered. There are several reasons for failures of sporadic operations. First, operational scripts often lack well-designed exception handling. Exception handling in infrastructure provisioning code is difficult, since it relies on different third-party systems. Second, cloud uncertainty[5], including both cloud resource failures and API call failures, contributes to the problem. Third, in DevOps release and deployment decisions are often delegated to many small teams, possibly causing version conflicts during system updates[1].

Existing recovery methods such as those for long-running transactions[33] have several drawbacks when applied on sporadic operations on cloud. First, it is hard to generalize them to cater for different types of sporadic operations, because recovery is usually application-specific and implemented in an intrusive way, like exception/error handling. Second, long-running transaction recovery needs to capture full checkpoints which represent the complete states of the system resources. This is inefficient and unsuitable for sporadic operations that only operate on a subset of the entire resources of a system. Another existing recovery method which has potential to do operational recovery is BPEL recovery[6]. BPEL recovery mechanisms treat system workflows as business processes and recover by taking compensating actions for any failures in system workflows[6]. However, BPEL’s recovery mechanisms usually deal with normal activities of cloud systems, and it requires a BPEL engine to execute[6]. Moreover, BPEL recovery has poor generalizability because its recovery installation is specific to a particular system workflow[6].

Given this context, we here propose a non-intrusive recovery framework, POD-Recovery, to recover from operational failures. The design of POD-Recovery is based on a set of requirements, which we derived from an analysis of relevant literature[7][8][9]. POD-Recovery is non-intrusive, meaning it requires no modification of the code performing the operations. Instead, it relies on the various logs the operation produces at runtime, and on access to cloud APIs for assessing the current state of resources and making changes. POD-Recovery also relies on being triggered by existing error detection and diagnosis frameworks, such as the open-source tools like POD-Diagnosis[10] or AFD[11].

POD-Recovery treats operations as processes consisting of logical steps. It therefore requires a training phase, which must be completed once per operation. The training phase determines the process as well as the set of cloud resources that are manipulated by the operation. It generates abstract templates of the expected states of these cloud resources after each step in the process, which are instantiated at runtime. The training phase further determines the recovery points
within the process, where checkpoints are taken at runtime. If failures are detected at runtime, the recovery service will either recover to a previous consistent state, or to the next expected state. These options result in a large set of possible pairs of current erroneous states and recovery goal states. Rather than statically encoding all possible recovery plans for such pairs, X-Recovery uses an AI planning technique[12] to dynamically generate recovery plans for a given situation.

We evaluate POD-Recovery against the set of recovery requirements over five representative types of sporadic operations on the AWS (Amazon Web Services) cloud[13]. To this end, we injected faults into 920 runs of those operations. The injected faults are based on interviews with industry and failure reports from practice, so that the errors are representative. The results show that POD-Recovery can recover from errors in sporadic operations in our experiments effectively, while satisfying the requirements.

The contributions of this paper are: 1) We propose and evaluate a novel and generalizable non-intrusive recovery method for sporadic operations on cloud. We provide a full discussion of the requirements of recovery and demonstrate how our recovery method is able to fulfill all the recovery requirements. 2) We propose a state management framework for operational recovery on cloud, comprising two components: resource space determination and expected resource states generation for a sporadic operation. 3) We demonstrate the feasibility of applying AI-planning techniques on recovery for cloud sporadic operations.

The rest of this paper is organized as follows. Section II introduces the background. Section III discusses the requirements for non-intrusive recovery. Section IV presents the design and implementation of our non-intrusive recovery service, which is evaluated in Section V. Threats to validity are discussed in Section VI, and related work in Section VII. Section VIII concludes the paper and points to future work.

II. BACKGROUND
A. Sporadic Operations on Cloud Applications

Sporadic operations on cloud applications are less frequently and regularly performed than normal cloud activities such as transactions on e-commerce websites. Sporadic operations usually include deployment activities such as installation, upgrade or migration of cloud systems[10][14][15]. Take the rolling upgrade operation as an example, its process model is shown in Fig. 1. Such a model is created by using log mining techniques[10]. The process model for rolling upgrade consists of seven steps. Steps 1 to 3 are sequential, and steps 4 to 7 are iterative. In steps 1 to 3, the new version of launch configuration (which specifies the images, security groups and other settings of newly launched instances) is created, and then the auto scaling group (which enables automatic scale in/out of instances) is updated to attach to this new version of launch configuration. In steps 4 to 7, the old instances (virtual machines) are deregistered from elastic load balancer and terminated, and the new instances will be launched and registered with elastic load balancer. Steps 4 to 7 are repeated until all the instances are upgraded.

Due to reasons such as cloud uncertainty[5], errors could occur in sporadic operations on cloud. For example, for step 5, the error of “old instance fails to be terminated” could happen, because this step remotely calls the cloud API of “TerminateInstancesInAutoScalingGroup” which might fail. Hence, we need to recover from these errors. In order to perform error recovery, we need an error detection framework. Several existing error detection frameworks for cloud operations exist, such as AFD[11], SFD[16] and POD-Diagnosis[10], which are able to detect runtime errors in cloud operations and able to trigger external recovery.

Fig. 1. Process Model for Rolling Upgrade.
B. Non-Intrusive Recovery VS. Intrusive Upgrade

Three possibilities exist for implementing a recovery method within any system: 1) it can be implemented without any information provided from the system; 2) it can be implemented using the information routinely provided by the system; or 3) it can be implemented by modifying the source code of the system. We call the first two options “non-intrusive recovery” since they do not involve modifying any code and the third option “intrusive recovery”. Our proposed recovery method in this paper is non-intrusive. The benefits of non-intrusive recovery over intrusive recovery include:

1) Non-intrusive recovery can handle errors from different systems in parallel by using a wider range of information such as logs from different systems and different monitoring facilities. In particular, for cases where successfully achieved conditions in previous steps are violated at later steps, non-intrusive recovery is able to handle them. For example, in step 2 of the rolling upgrade operation, if the resource generated in step 1 (Launch Configuration) is erroneous or was modified unexpectedly (e.g., the Launch Configuration was changed by another team) but was initially provisioned successfully, this error cannot be easily detected and handled by existing intrusive recovery mechanisms such as exception handling[17].

2) Non-intrusive recovery is independent of any programming language. The logs used to trigger error detection and recovery can be produced via any programming language.

3) It can be easily turned off, if needed: since it is non-intrusive, the recovery service can simply be shut down.

4) It spans tools. Many sporadic operations involve multiple tools used in a tool chain. Implementing an intrusive recovery method across multiple tools, for many of which the source code may not be available, is time consuming and
error prone, even if possible. By contrast, non-intrusive recovery is more convenient and feasible in such situations.

III. REQUIREMENTS FOR NON-INTRUSIVE RECOVERY

We derived six requirements for non-intrusive recovery, in part from the literature[7][8][9]. Our recovery framework is designed with the aim of satisfying all the requirements. We thoroughly analyzed these requirements and determined how each functional component in the recovery framework should be designed. Our experiments are structured to evaluate our recovery framework against these requirements.

A. R1: Runtime Recovery

In manual recovery during a sporadic operation, operators are usually alerted through some monitoring system and will then analyze logs and perform diagnostic tests to detect the causes. Meanwhile, the operation is likely to be stopped and rolled-back completely[18]. Hence, a requirement is that the recovery should be performed during the execution of the operation at runtime.

B. R2: Recovery Satisfying RTO

According to [7], RTO (Recovery Time Objective) refers to the goal of limiting the time for a recovery action to complete. For recovery of a hosted application itself during normal operation, RTO specifies the bound on how long it should take for an application to come back online after a failure occurs[7]. Our definition here is different. For recovery of a sporadic operation itself, RTO specifies an upper bound on how long it should take for a failed step to recover to either a previous consistent state or the expected state of the current step. Recovery time should not exceed the time boundary specified by RTO.

C. R3: Recovery from Errors without Known Causes

Recovery from errors during an operation may need information from both error detection and diagnosis. However, sometimes only the information from error detection is available. This is possible because 1) it may take too long to diagnose the error, or 2) the error was transient and there were no identifiable causes. Hence, a requirement is that the recovery should be able to recover from an error without knowing the cause of the error.

D. R4: Dealing with False Positives of Error Detection

Recovery is triggered after errors are detected by the error detection service. However, the detected errors might not be present. For example, during the operation an instance is stuck in the boot phase and this error is detected – but just after the error is discovered and before a recovery action is taken, the instance self-corrects. From the viewpoint of recovery, the detected error thus is a false positive. So, a requirement is that false positives of error detection should be caught and handled by the recovery.

E. R5: Recovery for Recovery Itself

Our recovery approach relies on a set of cloud API operations. Due to the uncertainty of those cloud API operations[5], the recovery service itself could fail. When the recovery service fails, the system may go into another erroneous state. Hence, ensuring the recovery from the failures of the recovery service itself is another requirement.

F. R6: Generalizability of Recovery

Many existing cloud recovery mechanisms[8][19][20][21] are either operation/application specific or for normal activities of applications. They usually depend on the contextual knowledge of the applications or operations. Applying them to different applications or operations will introduce additional effort. A requirement for our recovery service is that it can be applied to a different context with minimal effort.

IV. POD-RECOVERY: NON-INTRUSIVE RECOVERY FRAMEWORK

POD-Recovery is a non-intrusive recovery service, embodying the concept of “Recovery as a Service” (RaaS) [7][22][23]. POD-Recovery does not change any of the source code of the operation and is thereby non-intrusive. POD-Recovery is based on three assumptions: 1) cloud logs are accessible for error detection; 2) cloud APIs are accessible by external parties (e.g. cloud consumers or automation tools that can start and stop virtual machines, register them with load balancers, etc.); 3) cloud resources are accessible from external parties. Currently, all major public clouds fulfill those assumptions. For private clouds, it is subject to the management tools and their configuration whether those assumptions are met, in full or in parts.

POD-Recovery requires a process model of an operation, e.g. the process model of rolling upgrade introduced in Section II (Fig. 1) obtained by process mining[24]. The overview of POD-Recovery is illustrated in Fig. 2. Before an operation begins, we perform once-off off-line activities. First, we determine the operation resource space that governs the expected states and captured states during the operation being performed. The inputs are the mined process model together with the timestamps of each step of the process model, cloud API call logs which contain the cloud API call history from past successful runs of the operation (e.g. AWS CloudTrail logs[25]), and the API-Resource mapping table which specifies the cloud resources manipulated by each cloud API involved in the operation. Cloud API call logs list the API calls that took place, their timestamps, which API operation was called, parameters of relevant API calls, and the response information of each API call. The logs are automatically processed to identify the cloud resources being manipulated. Then using the information in cloud API call logs such as AWS CloudTrail logs[25], we automatically generate a template for the expected state of the affected resources at each process step. We do this by correlating API calls with steps in the process model using timestamps. The templates include variables, e.g. for resource IDs, whose values need to be obtained at operation runtime. Finally, the recovery points inside an operational process are determined. The failures occurring within the operation step(s) between two adjacent recovery points are recoverable. At runtime, the external error detection service observes the cloud operation, e.g. through its logs, and invokes the recovery service if errors are detected. The recovery service then obtains the current state of the relevant cloud resources. This current state of these resources can be more recent than the erroneous state identified by the error detection service so it effectively serves as a double check to reduce false positives from error detection. The service-oriented design has two benefits here: 1) the recovery service can be implemented in any programming language; 2) the recovery service is more generalizable and easy to maintain since it is independent of the error detection service. POD-Recovery implements two recovery patterns: Rewind & Replay and Reparation. Rewind & Replay brings the system resources back to the saved
consistent resource state before re-executing a step. Reparation brings the system resources to a desired resource state directly from the erroneous state.

---

**Generalized Recovery Workflow**

Fig. 3 describes the detailed recovery workflow. Before the operation starts, the off-line activities are performed. Say, an error is detected after a recovery point (e.g. Step Section 1). Then the current erroneous resource state is captured (e.g. Resource State 1). The recovery service also retrieves the expected resource state template and sets the variable values from operation logs (e.g. Expected State 1). The recovery service retrieves a previous consistent resource state (e.g. Resource State 0). Then recovery is executed by using these three states. Rewind & Replay requires the previous consistent resource state and the current erroneous resource state, and Reparation requires the expected resource state and the erroneous resource state. After recovery, the latest resource state (e.g. Resource State 1) will be captured again before continuing.

**1) Determining Operation Resource Space**

The operation resource space is the set of manipulated resources during all steps of an operation. The key inputs needed for determining the resource space are cloud API call logs, which contain all the API operations that were called in past operation runs. Different cloud platforms offer this functionality. For instance, on AWS the respective service is called CloudTrail[25]. Fig. 4 shows an example of AWS CloudTrail logs in JSON format: a record of a call to “UpdateAutoScalingGroup”. It specifies the API call time (“eventTime”), operation (“eventName”), call parameters (“requestParameters”) and response (“responseElements”).

---

**Generating the resource space relies on the information provided by logs like CloudTrail.** The approach for generating the resource space is illustrated in Fig. 5. First, based on timestamped logs of an operation process and on CloudTrail logs, the operation-related API calls can be determined by correlating the timestamps in operation process and in cloud trail logs. Second, based on the API-resource mapping table we can obtain the overall set of resources changed by the operation. The mapping table lists which resources are changed by which API operations – e.g., the operation “createLaunchConfiguration” changes resources of type “Launch Configuration”. We created this mapping table manually, which is feasible for the limited number of resource types and APIs in typical clouds.

---

**Using rolling upgrade as an example, the resource space contains four resource types: Instance (i.e. VM), Launch Configuration (LC), Auto Scaling Group (ASG), and Elastic Load Balancer (ELB). Each resource type has attributes (e.g. instances have an instance id, instance type, machine image, etc.). The dependency relationships among those resources are: 1) an ASG is associated with one LC; 2) an ASG contains a (possibly empty) set of instances; 3) an ELB links to a (possibly empty) set of instances; 4) each instance belongs to at most one ASG; and 5) each instance can be linked to a (possibly empty) set of ELBs. These relationships also align with the resources dependency relationship specified by AWS documentation[26]. Determined, we use the resource space as an input to the resource state capturing service. This service is responsible for capturing the states of the resources of the target cloud system.
2) Generating Expected Resource State Templates

While the resource state determination captures the affected resources on the level of a whole process, our approach requires a finer level of detail: for each step in the process, the expected resource state templates capture which resources are affected how. These templates are also generated from historic logs of the operation.

---

![Figure 6. Expected Resource State Templates Generation.](image)

Fig. 6. Expected Resource State Templates Generation.

---

In the generated resource state templates, most fields have variable content. These variables are represented with question marks as prefix (e.g. “?newLC”) in Fig. 7. For the set of instances, it is unclear at design time how many instances will be in the initial state – symbolized by “ […]” in Fig. 7. At runtime, the actual values for variables, the number of instances, etc., need to be set. All of the fields in the initial state can be populated when the operation starts, by correlating the CloudTrail log entries’ timestamps with the operational steps’ timestamps. Then we derive the initial resource state of the system before the operation starts, by reasoning over the resource state space, the affected resource types, and their attributes. An example of such an initial state is shown, in XML, as the top part of Fig. 7. Next, based on API call 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 result of applying the API operations from state 1 to the initial state (e.g. a new LC is created), the expected state after step 2 is the result of applying the API operations from step 2 to the state after step 1 (e.g. ASG is updated), and so on. The bottom part of Fig. 7 shows an example of the expected resource state templates after step 1 (Create New LC) of rolling upgrade. When the operation starts, the resources are in the initial state. Then, after calling the API operation “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”. Each generated resource state template follows the structure of the determined resource space.

3) Recovery Points Determination

We define recovery points in an operational process to be the positions in the operation where the presence of errors should be checked and, if needed, recovery should be triggered. A recovery point thus also doubles as a consistent checkpoint: if an error is detected in the current recovery point, the previous recovery point is the last consistent checkpoint. As such, we only recover from errors that occurred since the last recovery point. Recovery points are manually determined. To decide where to place recovery points, we adopt a criterion, called Recovery Actions Identifiable, from the literature[14]. By applying this criterion, the operation can be divided into several recoverable sections, where each section ends with a recovery point.

---

![Figure 7. Sample Expected Resource State Templates.](image)

---

![Figure 8. Determining Recovery Points and respective sections.](image)
Fig. 8 shows the sections resulting from recovery points at different levels of granularity. First, Fig. 8(a) is obtained by applying Recovery Actions Identifiable to the steps of rolling upgrade. Since each step has at least one recovery action, each step forms a section and there are seven sections. Fig. 8(b) combines some sections with low failure rates. According to the literature [27], activities such as “creating launch configuration” and “update auto scaling group” have low failure rates whereas activities such as “launch instances” and “terminate instances” have relatively high failure rates. Therefore, the first three sections from Fig. 8(a) can be combined into one section. After each of the remaining five sections there is a recovery point. When needed, recovery can take place after each recovery point.

B. Online Activities

Using the process model obtained by processing mining and the models generated offline, we describe how online recovery works in our framework. In particular, mining the process model is independent of the offline and online activities of POD-Recovery, and hence it does not cause any performance overhead on POD-Recovery. Process mining can ensure the accuracy of the mined process model with respect to its input, i.e., the logs from which the model is mined.

1) Capturing System Resource States

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 [28], our state capturing method only captures the states of the resources involved in the determined resource space. This increases the efficiency of state capture, especially when the system scale is large. The state capturing algorithm captures relevant state information of each resource in the resource space. For each cloud resource, it retrieves the current state through calls to public cloud APIs. For AWS, this includes calling operations such as “DescribeInstances” or “DescribeLoadBalancer” and recording the results. The state capturing service is triggered to capture the initial state at each recovery point. Also, if recovery takes place for some recovery point, the resource state is captured after recovery completes.

2) Recovery Patterns

Out of a set of eight conceptual recovery patterns proposed in our earlier work [15], we adopt and implement two recovery patterns here: 1) Rewind & Replay and 2) Reparation. It should be noted that the approach is extensible to cover more recovery patterns, but in this paper we focus on building a complete framework around the two patterns above. We selected these two patterns as they resemble common forward and backward recovery strategies.

Fig. 9 shows the mechanics of these two recovery patterns. Before the current Section X, state CS0 was captured. If Section X executes correctly, the system reaches expected state ES1. But errors may occur in Section X, leading the system into an erroneous state S_err. The two recovery patterns offer alternative ways to recover from the errors. In Rewind & Replay, the goal is to bring the system first from S_err back into CS0, and then to re-execute Section X. In contrast, the goal of Reparation is to bring the system directly from S_err to ES1. The resource states can be captured by using the resource state capturing service described above, and the expected resource state templates can be generated by the expected states generation algorithm described in Section IV.A.2). The variables in the resource state templates are set to values derived from runtime logs of the operation. How to achieve the state transitions, i.e. from S_err to CS0 for Rewind & Replay and from S_err to ES1 for Reparation, is discussed below. This is achieved through automated planning, which generates recovery plans from S_err to the desired goal state.

The algorithms for the two recovery strategies are shown in Fig. 10. Rewind & Replay in Fig. 10(a) starts by generating a recovery plan from the current erroneous state S_err to the previous consistent state CS0. As discussed in more detail in the next section, there may be cases where no plan is found. If a plan is found, the generated plan is executed. Then, Section X is re-executed. The algorithm for Reparation, Fig. 10(b), follows a similar logic – however, the recovery plan brings the system from the erroneous state S_err directly to the expected state ES1, therefore no re-execution of the original steps is needed. If the recovery attempt fails due to errors occurring during the execution of the recovery plan, the exception handler captures the current resource state of the system and recursively calls the same recovery algorithm, respectively.

3) Recovery Plan Generation using AI Planning

The set of possibly needed recovery plans in general is large, comprising all pairs (S_curr, CSX) and (S_curr, ESX). Depending on the actual operation, the set of possible error states S_curr may be infinite. Therefore, any solution that requires pre-specified recovery plans would be limited, and any requirement on pre-specifying recovery plans increases the burden for adopting the recovery approach. Furthermore, the state of a cloud application cannot be seen as memory state: recreating resources like running VMs is categorically different from reverting back to a checkpoint or a snapshot of a database. For these reasons, we adopted an automated planning approach for the generation of recovery plans. Previous research on undo of cloud operations [29][37]...
successfully used artificial intelligence (AI) planning to move a cloud system from a given state to an earlier consistent state, in order to undo undesired changes. The AI planner used is Fast-Forward (FF) in the variant described in [12]. It requires three types of inputs: an initial state, a goal state, and a set of action templates referred to as the planning domain. The actions in the planning domain are specified in the Planning Domain Definition Language (PDDL) [12]. Each action’s specification comprises its preconditions, its parameters, and its effects. From these inputs, the planner generates a state transition plan.

We adopted this approach for recovery plan generation in POD-Recovery. The undo tool [37] is publicly available, and includes a domain model for a number of the AWS cloud API operations. While this facilitated re-use significantly, we had to add full support for certain cloud resource types such as auto scaling groups and elastic load balancers. For the eight recovery patterns from our earlier work [15], we analysed the possibility to map all eight patterns to AI planning tasks. This mapping is illustrated in Table I. As can be seen, six of the eight patterns can, at least in part, be mapped to AI planning problems. The multitude of different pairs of planning tasks further underlines our choice of automated planning for recovery. In this paper, however, we specifically focus on two recovery patterns as explained above. We therefore consider two types of resource state transitions in our current recovery method: 1) from the erroneous state to the last captured state (Rewind & Replay) and 2) from the erroneous state to the expected state (Reparation).

### TABLE I. AI-PLANNING FOR EIGHT RECOVERY PATTERNS

<table>
<thead>
<tr>
<th>Recovery Pattern</th>
<th>AI-Planning Recovery Plan</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compensated Undo &amp; Redo</td>
<td>From current erroneous state to expected state before the step ($S_{curr}, ES(X)$)</td>
</tr>
<tr>
<td>Compensated Undo &amp; Alternative</td>
<td>From current erroneous state to expected state before the step ($S_{curr}, ES(X)$)</td>
</tr>
<tr>
<td>Rewind &amp; Replay</td>
<td>From current erroneous state to captured state before the step ($S_{curr}, CS(X)$)</td>
</tr>
<tr>
<td>Rewind &amp; Alternative</td>
<td>From current erroneous state to captured state before the step ($S_{curr}, CS(X)$)</td>
</tr>
<tr>
<td>Reparation</td>
<td>From current erroneous state to expected state after the step ($S_{curr}, ES(X)$)</td>
</tr>
<tr>
<td>Direct Redo</td>
<td>N.A.</td>
</tr>
<tr>
<td>Direct Alternative</td>
<td>N.A.</td>
</tr>
<tr>
<td>Farther Undo &amp; Redo</td>
<td>From current erroneous state to expected state before the step prior to last ($S_{curr}, CS(X)−2$)</td>
</tr>
</tbody>
</table>

How we embed AI planning for generating recovery plans is illustrated in Fig. 11. First, the initial state (captured current error state) and the goal state (either a previous captured state or an expected state) are captured/derived, and translated into a planning problem in PDDL. Second, with both the state transition problem file and the planning domain model file as inputs, the AI planner is called to automatically generate the recovery plan. In some cases, this is not possible: certain actions, like deleting virtual hard disks, are not reversible unless additional precautions are taken. A detailed analysis of the undoability of AWS API operations can be found in [29]. If the planner determines that no recovery plan exists, the respective recovery pattern is not applicable to the current erroneous state. Otherwise, the recovery plan achieves resource state transitions from the initial state to the goal state. Our extended planning domain model file has more than 1300 lines of PDDL code, which cover all the possible AWS API operations that can affect the resource space. The recovery plan generated by the FF planner variant [12] typically is the shortest path, i.e., the least number of actions needed. Finally, since the generated plan is returned in a proprietary format, it needs to be translated to executable code, like command line scripts. The translated code is then executed, and calls API operations like “RunInstances”. Once it completes, the system is in the desired goal state — unless errors occurred, in which case a recursive call to the recovery procedure is made. Since the recovery plans are dynamically generated from the current error state, the AI planner can generate recovery plans for failures that occur during recovery itself.

![Fig. 11. AI-Planning based Recovery Plan Generation.](image)

One challenge lies in ensuring the efficiency of recovery plan generation: automated planning is a computationally hard problem [12]. Our resource space determination mechanism is helpful in this regard, by not considering cloud resources that are unrelated to the current operation. If unrelated resources are present in the planning problem, the planner has to consider them during each intermediate planning state, which makes the search for a plan more time-consuming. Another challenge was that the planning task generation in POD-Recovery needs to explicitly state if any resources should be deleted — else the planner does not include deletion actions. For example, if the initial state has three instances (a, b, c) and the goal state has two instances (a, b), the semantics used in the planner imply that there is no need to change anything, and hence the planner returns an empty plan. We solve this problem by explicitly adding the keyword “Del” to the deleted resources in the goal state (e.g. Del(c)). This matches the planning domain, where the effect of any “delete” action is that the resource will be deleted (Del(resource)). By making it explicit which resources need to be deleted, the planner can select the respective actions.

V. EXPERIMENTS & EVALUATION

In this section, we discuss the experiments we performed to test POD-Recovery on AWS EC2 [13]. Our techniques can also be adapted to other cloud platforms as long as they can meet the three aforementioned assumptions: 1) cloud logs are externally accessible; 2) cloud APIs are externally accessible; 3) cloud resources are externally accessible. We begin by discussing the experimental environment. Then we describe the particular experiments to test the recovery requirements outlined in Section III. For most requirements, we describe the experimental procedures, the experimental results, and discuss how well POD-Recovery satisfies the respective requirement. Our experiments are largely based on running an operation and injecting cloud faults into a run, e.g. by calling the cloud API to terminate a VM or to change the configuration of a load balancer. As such we inject faults into the system, but because these are immediately activated, they become errors directly. They may or may not lead to user-perceivable failures, directly or eventually, but we focus on recovering from errors.
Our approach is novel and categorically different from existing solutions, in that it is both non-intrusive and under the control of the cloud customer. These differences render any comparative evaluation unreasonable. For instance, comparing POD-Recovery with self-healing mechanisms provided by clouds themselves is not meaningful: these mechanisms are not under the customer’s control, and exist regardless of whether POD-Recovery is used.

A. Experimental Environment

The experimental environment is shown in Fig. 12. The cloud application we use is comprised of several stateless web servers attached to an auto scaling group and registered in an elastic load balancer. The operation tool, Asgard, runs as a Web app in a dedicated server, with its client side running on the operator’s machine. The operator interacts with the cloud environment and the associated resources through Asgard. Logs generated by Asgard and the cloud are collected by the LogStash service[30], running in a LogStash server. The error detection service also runs in a dedicated server, and it relies on the logs collected by LogStash for detecting errors at the recovery points within the operation. POD-Recovery runs in the recovery server. The recovery service is triggered by the error detection service[10] to do recovery.

![Experimental Environment](image)

Fig. 12. Experimental Environment.

B. Runtime Recovery (Satisfying Recovery Requirement R1)

As should be clear from the description in Section IV, our method is inherently a runtime recovery solution. Since POD-Recovery is non-intrusive, it has no means of controlling the operation process itself. Asgard has generous timeouts, as can be seen from its source code or its logs – in some cases up to 70 minutes. As shown in the experiments for the other requirements, POD-Recovery often perform all needed recovery actions within one minute. Even in the worst case tested, recovery completed within 16 minutes, well before the timeouts are reached. We discuss threats to validity, such as these favourable conditions, in Section VI.

C. Recovery Satisfying RTO

1) Experimental Procedure

This requirement concerns the time it takes recovery the system to a consistent state. We therefore measure recovery times by performing rolling upgrade on a cloud application supported by 12 instances. We vary the rolling depth parameter – which controls how many VMs are upgraded in parallel – over the values of 1, 2, 3, 4, and 6. We used our own proprietary fault injection tool to inject faults for each of the five determined recovery points in rolling depth setting, and performed recovery according to either of the two strategies. Per rolling depth setting, we ran the experiment 30 times, measuring the recovery times. The faults injected are described in Table II. For some recovery points, multiple types of faults can be injected. For example, for recovery point 3 with rolling depth of 6, the resulting errors could be that 1 instance is not terminated, 2 instances are not etc. The selection of injected faults is based on our interviews with industry and failure reports, and hence representative.

![Recovery Time of Rewind & Replay](image)

Fig. 13. Recovery Time of Rewind & Replay.

![Recovery Time of Reparation](image)

Fig. 14. Recovery Time of Reparation.

2) Experimental Results

Fig. 13 shows the recovery time when using the Rewind & Replay strategy, and Fig. 14 for the Reparation strategy. Recovery time here refers to exclusively the time for executing the recovery plan. In both figures we only show the recovery time for the worst fault configuration – e.g. for rolling depth of 6 we typically show times for the fault of 6 instances being terminated, since it takes the longest to recover from. All the other configurations have recovery times no worse than the cases shown. The recovery time in the figures represents the average value of the recovery time for 30 runs for the respective configuration. The highest relative standard deviation for any configuration and any strategy is 2.4% (Rewind & Replay, rolling depth of 6, at recovery point 4).

<table>
<thead>
<tr>
<th>Recovery Point</th>
<th>Fault injected</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recovery Point 1 (After Step 3)</td>
<td>ASG uses unknown LC</td>
</tr>
<tr>
<td>Recovery Point 2 (After Step 4)</td>
<td>Instance still registered with ELB</td>
</tr>
<tr>
<td>Recovery Point 3 (After Step 5)</td>
<td>Instance not terminated</td>
</tr>
<tr>
<td>Recovery Point 4 (After Step 6)</td>
<td>Instance launching fails</td>
</tr>
<tr>
<td>Recovery Point 5 (After Step 7)</td>
<td>Instance not registered with ELB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Recovery</th>
<th>Table II. FAULTS INJECTED FOR ROLLING UPGRADE OPERATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recovery Point 1</td>
<td>ASG uses unknown LC</td>
</tr>
<tr>
<td>Recovery Point 2</td>
<td>Instance still registered with ELB</td>
</tr>
<tr>
<td>Recovery Point 3</td>
<td>Instance not terminated</td>
</tr>
<tr>
<td>Recovery Point 4</td>
<td>Instance launching fails</td>
</tr>
<tr>
<td>Recovery Point 5</td>
<td>Instance not registered with ELB</td>
</tr>
</tbody>
</table>

According to [31], RTO (Recovery Time Objective) can be achieved by adopting appropriate recovery actions. In the
domain of operations on cloud, we adopt two appropriate recovery patterns: 1) Rewind & Replay and 2) Reparation. From the results we can see that for Rewind & Replay recovery time is at most 35 seconds (s), and at most 18s for Reparation. In addition, for these experiments state capturing time was typically 14s but at most 15s, and recovery plan generation time was at most 50 milliseconds (ms). While not part of our scope here directly, error detection time has to be considered when discussing RTO as well. Detection times reported in the literature ranged from 10ms to 11 seconds[10].

The sum of the worst cases of all the above times, i.e. the total error detection and recovery time, is around 61s. For comparison, we inspected the Asgard source code to find the original timeout values implemented for each of the five recovery points. These are 10 minutes (min), 10min, 70min, 50min, and 10min, respectively. If such times are acceptable, 61s in the worst case is a steep improvement – unless errors repeatedly occur during the recovery. Hence, either of the recovery strategies achieves a reasonable RTO.

D. Recovery from Errors without Known Causes

1) Experimental Procedure

POD-Recovery does not require a known cause: its recovery plan generation dynamically finds a plan from the current erroneous state to a desired state – be it a captured earlier state or an expected future state. To study this aspect in detail, we performed additional experiments with other faults. We again ran rolling upgrade, injected various faults, and triggered recovery without providing the error causes. Per type of injected fault, we performed 30 runs.

2) Experimental Results

Table III summarizes the results of these experiments, by showing the number of successful recoveries for each injected fault. As can be seen, recovery is successful for all injected faults, except for one: the fault “cloud system disconnected” refers to a network disconnect between the recovery service and the AWS API. Due to this disconnect, POD-Recovery cannot take any recovery actions or even assess the current state of the resources, and hence cannot recover.

<table>
<thead>
<tr>
<th>Recovery Point</th>
<th>Error Occurred</th>
<th>Fault Injected (Cause)</th>
<th># Successful Recoveries</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recovery Point 1</td>
<td>ASG uses wrong launch configuration</td>
<td>Launch configuration (LC) changed by other team</td>
<td>30</td>
</tr>
<tr>
<td>Recovery Point 2</td>
<td>Old instance cannot be deregistered from ELB</td>
<td>“DeregisterInstancesFromELB” call fails</td>
<td>30</td>
</tr>
<tr>
<td>Recovery Point 3</td>
<td>Old instance cannot be terminated</td>
<td>Other team reattaches an old instance to ELB</td>
<td>30</td>
</tr>
<tr>
<td>Recovery Point 4</td>
<td>Auto scaling group has wrong version instances</td>
<td>Cloud system disconnected</td>
<td>0</td>
</tr>
<tr>
<td>Recovery Point 5</td>
<td>New instance cannot be registered with ELB</td>
<td>Other team detaches this new instance from ELB</td>
<td>30</td>
</tr>
</tbody>
</table>

3) Satisfying Recovery Requirement R3

Due to the dynamic recovery plan generation, our method can recover from errors without known cause – as long as the error cause is included in the resource space (e.g. wrong LC) and can be recovered through API calls. If the cause is not included or the API cannot be reached (e.g. cloud system disconnected), our method cannot recover from the error. Hence, the cause of the error does not affect the recovery although it may affect whether the error remains or reoccurs.

E. Dealing with False Positives of Error Detection

1) Experimental Procedure

To test the reaction of POD-Recovery to false positives, we trigger the recovery service without injecting an error, and observe whether POD-Recovery takes any action. For each recovery point (RP) we do this 50 times. Since error detection triggers the recovery service, and since POD-Recovery does not consider any causes offered by the triggering message, we do not investigate any more complex false positives. Also, as argued above, if a transient error disappeared before recovery started, we consider it to be a false positive from the viewpoint of recovery.

2) Experimental Results

Table IV shows the recovery service’s ability to handle false positives from the error detection service. We can see that all the false positives were successfully detected: in all 250 runs of our experiment, no recovery action was taken.

<table>
<thead>
<tr>
<th>Recovery Point</th>
<th># False Positives</th>
<th># No Recovery</th>
</tr>
</thead>
<tbody>
<tr>
<td>RP 1</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>RP 2</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>RP 3</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>RP 4</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>RP 5</td>
<td>50</td>
<td>50</td>
</tr>
</tbody>
</table>

3) Satisfying Recovery Requirement R4

Due to the design of our recovery method, with its dynamic plan generation, false positives from error detection will not cause unnecessary recovery actions.

F. Recovery for Recovery Itself

1) Experimental Procedure

To test POD-Recovery’s behavior when errors occur during the execution of the recovery plan, we performed a rolling upgrade operation on a cloud application with 12 instances, at a rolling depth of 1. Hence, during the rolling upgrade, 1 old instance will be killed and 1 new instance will be launched. We injected faults into each determined recovery point to trigger our error recovery service, waited for the recovery plan to execute, and manually injected additional faults then. We performed 30 runs of this test of “recovery for recovery itself” for each of the two recovery strategies and each recovery point.

2) Experimental Results

Table V shows the recovery time of recovery for the two recovery strategies on rolling upgrade operation. The relative standard deviation is at most 2.3% (“Recovery for Reparation”, at recovery point 4).

<table>
<thead>
<tr>
<th>Recovery Time of “RECOVER FOR RECOVERY”</th>
</tr>
</thead>
</table>
---|---|---
RP 1 | 8.23 s | 3.51 s
RP 2 | 4.89 s | 3.89 s
RP 3 | 24.9 s | 9.91 s
RP 4 | 35.83 s | 10.78 s
RP 5 | 5.67 s | 4.98 s

3) Satisfying Recovery Requirement R5
We can see that the maximum recovery time of “recovery for Rewind & Replay” is 35.83 seconds (at recovery point 4), and the maximum recovery time of “recovery for Reparation” is 10.78 seconds (at recovery point 4). Hence, the times for state capturing, plan generation, the original recovery, and the “recover for recovery” in total is much less than the specified time gap between two adjacent recovery points. Thus, the requirement of “recovering for recovery” is achieved by our recovery framework.

### G. Generalizability of Recovery

#### 1) Experimental Procedure
To test generalizability, we applied our method to 4 other operations for cloud applications: 1) scale-up; 2) scale-down; 3) installation; and 4) migration. Those 4 operations plus the rolling upgrade operation itself are a good cross section of sporadic operations on the cloud.

The scale-up operation has 3 steps: 1) increase auto scaling group’s desired capacity by a given number n; 2) wait for the auto scaling group to start the additional n instances; 3) register these n instances with the load balancer. The recovery points are determined to be after step 2 and step 3 based on the recovery point determination criterion and failure rate consideration. We injected faults for the two recovery points and measured the recovery time of the two recovery strategies based on 30 runs. The faults injected are described in Table VI. We tested three scale-up settings: 1) from 12 to 24 instances; 2) from 12 to 36 instances; and 3) from 12 to 100 instances.

#### TABLE VI. FAULTS INJECTED FOR SCALE-UP

<table>
<thead>
<tr>
<th>Recovery Point</th>
<th>Fault injected</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recovery Point 1 (After Step 2)</td>
<td>25% instances with wrong AMI</td>
</tr>
<tr>
<td>Recovery Point 2 (After Step 3)</td>
<td>25% instances not in ELB</td>
</tr>
</tbody>
</table>

The scale-down operation also has 3 steps: 1) decrease auto scaling group’s desired capacity by a given number n; 2) deregister n instances from the elastic load balancer; 3) wait for the auto scaling group to terminate n instances. The recovery points are determined to be after step 2 and step 3 based on the recovery point determination criterion and failure rate consideration. We injected faults for the two recovery points and measured the recovery time of the two recovery strategies based on 30 runs. The faults injected are described in Table VII. We tested the three scale-down settings that are the inverse of the scale-up settings above: from 24 to 12 instances, etc.

#### TABLE VII. FAULTS INJECTED FOR SCALE-DOWN

<table>
<thead>
<tr>
<th>Recovery Point</th>
<th>Fault injected</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recovery Point 1 (After Step 2)</td>
<td>n/4 instances still in ELB</td>
</tr>
<tr>
<td>Recovery Point 2 (After Step 3)</td>
<td>n/4 instances not terminated</td>
</tr>
</tbody>
</table>

The installation operation has 4 steps: 1) launch a web instance; 2) assign an elastic IP to the web instance; 3) launch a database instance; 4) assign an elastic IP to the database instance. Based on our procedure, the recovery points were determined to be after step 2 and step 4. Once again, we injected faults for the two recovery points and measured the recovery time of the two recovery strategies based on 30 runs. The faults injected are described in Table VIII.

#### TABLE VIII. FAULTS INJECTED FOR INSTALLATION

<table>
<thead>
<tr>
<th>Recovery Point</th>
<th>Fault injected</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recovery Point 1 (After Step 2)</td>
<td>Web instance not launched</td>
</tr>
<tr>
<td>Recovery Point 2 (After Step 4)</td>
<td>Database instance not launched</td>
</tr>
</tbody>
</table>

The migration operation has 10 steps: 1) stop the old web instance in the old availability zone; 2) create an AMI from the old web instance; 3) launch a new web instance in the new availability zone, using the AMI; 4) reassign the IP to the new web instance; 5) terminate the old web instance; steps 6) – 10): repeat the above for the database instance. The recovery points are determined to be after steps 1, 2, 4, 5, 6, 7, 9, and 10, based on our procedure. We injected faults as per Table IX for the eight recovery points and measured the recovery time of the two recovery strategies based on 30 runs.

#### TABLE IX. FAULTS INJECTED FOR MIGRATION

<table>
<thead>
<tr>
<th>Recovery Point</th>
<th>Fault injected</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recovery Point 1 (After Step 1)</td>
<td>Old web instance not stopped</td>
</tr>
<tr>
<td>Recovery Point 2 (After Step 2)</td>
<td>AMI not created</td>
</tr>
<tr>
<td>Recovery Point 3 (After Step 4)</td>
<td>New web instance not launched</td>
</tr>
<tr>
<td>Recovery Point 4 (After Step 5)</td>
<td>Old web instance not terminated</td>
</tr>
<tr>
<td>Recovery Point 5 (After Step 6)</td>
<td>Old DB instance not stopped</td>
</tr>
<tr>
<td>Recovery Point 6 (After Step 7)</td>
<td>AMI not created</td>
</tr>
<tr>
<td>Recovery Point 7 (After Step 9)</td>
<td>New DB instance not launched</td>
</tr>
<tr>
<td>Recovery Point 8 (After Step 10)</td>
<td>Old DB instance not terminated</td>
</tr>
</tbody>
</table>

2) Experimental Results
Figures 15-18 show the average recovery times of the two recovery strategies for each recovery point in the four different operations, respectively. Fig. 15 depicts these times for scale-up. There are three scale-up settings: from 12 to 24 instances, from 12 to 36 instances, and from 12 to 100 instances. The recovery time in this figure represents the average of the recovery time for 30 runs. The highest relative standard deviation was 3.4% (from 12 to 100 instances, at recovery point 1, Rewind & Replay).

![Recovery Time for Scale-up Operation](image-url)
Fig. 16 shows the average recovery times for the scale-down operation and both recovery strategies for each recovery point. The highest observed relative standard deviation was 2.9% (from 100 to 12 instances at recovery point 2, using Rewind & Replay).

![Fig. 16. Recovery Time for Scale-down Operation.](image)

Fig. 17 shows the average recovery time of the two recovery strategies for each recovery point in the installation operation. The relative standard deviation was at most 2.7% (at recovery point 2, Rewind & Replay).

![Fig. 17. Recovery Time for Installation Operation.](image)

Fig. 18 shows the recovery time of the two recovery strategies for each recovery point in the migration operation, averaged over the 30 runs. The highest relative standard deviation is 3.5% (Rewind & Replay, at recovery point 7).

![Fig. 18. Recovery Time for Migration Operation.](image)

3) Satisfying Recovery Requirement R6

The results show that our method is also able to recover for other sporadic operations such as scale up/down operations, installation and migration. Since these operations cover a wide cross section of cloud operations, there is strong indication that our recovery framework is generalizable.

VI. THREATS TO VALIDITY

First, we address the arguably favorable conditions for our experiments. All major public clouds offer management APIs and respective call logs that can be used for our purposes, so the related assumptions apply to a lot of circumstances. However, Netflix Asgard provides logs of reasonably good quality, which contain all information needed by our approach.

Second, the once-off effort of generating the needed models may be considered high in a given context. However, typically not many different operations are executed very frequently, so the effort can be focused on those. For a particular operation and context, a relatively large amount of work is needed to generate the resource space and expected states from historical data of past operation runs. The data must be representative of the operations and systems for which POD-Recovery is set up.

Third, we test the generalizability of our method by using five representative sporadic operations. Since sporadic operations on cloud involve more than these five types of operations, more test towards the generalizability of our recovery methodology would be desirable.

Fourth, we argue that our recovery framework can also be applied on other cloud platforms such as Windows Azure[32], as long as the cloud resources, cloud logs, operation logs, and cloud management APIs are available. Also, if a suitable domain model is provided to the AI planner, it can generate recovery plans for other clouds as well. While we have created a partial AI planning domain model for OpenStack, we have not conducted full tests regarding the applicability of POD-Recovery to other clouds.

VII. RELATED WORK

Our work relates to test driven scripting, recovery for long-running transactions, and log analysis as follows.

A. Test Driven Scripting

OpsCode[35] scripts such as Chef[36] can be used for implementing consumer-initiated cloud sporadic operations. In order for the scripts to be more reliable, operation scripts can be written in a test driven manner[19], e.g. by using script mini tests[19]. Specifically, mini tests usually test the functionality and availability of a module in the whole script infrastructure. Mini tests are carried out over a test bed which is different from the production environment, and can test the behavior for a known set of error conditions through error handling facilities. In contrast, our recovery service offers a generic framework for handling known and unknown error conditions in production.

B. Recovery within Long-Running Transactions

For long running transactions, recovery strategies usually involve backward or forward recovery[33]. Backward recovery refers to the strategy which first reverts the current erroneous state to a previous correct state before attempting to continue execution. Forward recovery attempts to correct the current erroneous state and then continues normal
execution. Another form of forward recovery is compensation[33], which means to attempt to correct the state of a system given some knowledge of the previous actions of the system[33]. Our Rewind & Repay is inspired from backward recovery and our Reparation is inspired from forward recovery.

C. Log Analysis for Error Detection & Diagnosis

Log analysis is a common technique for assisting in the detection and diagnosis of errors, see e.g. [34]. Our work differs from prior works in that we create a process model to provide context for the logs and we use the logs online to correct errors rather than offline to detect errors. The logs are analysed during the execution of operations at runtime, rather than after the operations are terminated. Moreover, we rely on logs to trigger the error detection service, which further triggers our non-intrusive recovery service.

VIII. CONCLUSION & FUTURE WORK

For reasons like cloud uncertainty, sporadic operations on cloud can fail. Especially with more and more organizations adopting DevOps and continuous deployment methodologies, this becomes a major risk. We therefore proposed and evaluated a non-intrusive recovery method for sporadic operations on cloud, called POD-Recovery. In order to successfully apply POD-Recovery, we need access to operation logs, cloud API call logs, and the cloud APIs. From the logs, we create an accurate process model of the operation, a model of the cloud resources manipulated by the operation and a model of the expected resource state templates of the steps of the operation. With these models and a domain model of the actions in the cloud APIs, we can dynamically generate recovery plans for a current erroneous state by employing an AI planner. The experimental results showed that POD-Recovery can recover from operational errors effectively while by and large satisfying all recovery requirements. The open questions for our future work are: 1) to which degree do our assumptions and conditions hold for other domains and cloud services, and 2) how can the assumptions be relaxed so that non-intrusive recovery systems can be more broadly applied.

ACKNOWLEDGMENTS

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


[34] Logstash Official Website: http://logstash.net/ (last access time: 15 Apr 2016, 13:50).

