# Automatic Undo for Cloud Management via AI Planning

Ingo Weber[1,2], Hiroshi Wada[1,2], Alan Fekete[1,3], Anna Liu[1,2], and Len Bass[1,2]

[1]NICTA, Sydney
[2]School of Computer Science and Engineering, University of New South Wales
[3]School of Information Technologies, University of Sydney
[1]{*firstname.lastname*}*@nicta.com.au*

## 1 Introduction

The facility to rollback a collection of changes, i.e., reverting to a previous acceptable state, a *checkpoint*, is widely recognised as valuable support for dependability [3, 5, 10]. This paper considers the particular needs of users of cloud computing resources, wishing to manage the resources. Cloud computing provides infrastructure programmatically managed through a fixed set of simple system administration commands. For instance, creating and configuring a virtualized Web server on Amazon Web services (AWS) can be done with a few calls to operations that are offered through the AWS management API. This improves the efficiency of system operations; but having simple powerful system operations may increase the chances of human-induced faults, which play a large role in overall dependability [24, 25]. Catastrophic errors, like deleting a disk volume in a production environment, can happen easily with a few wrong API calls.

To support dependability in a cloud platform, it would be helpful if the platform made it easy for a user to rollback to recover from failure. However, the nature of a cloud platform introduces particular difficulties for this approach. The user cannot alter the set of operations provided in a management API, nor can he tailor or even examine its implementation – users have to accept a given API, which is not necessarily designed to support undo. Thus, restoring a previous acceptable state can only be achieved by choosing an appropriate set of API operations, and calling them in a particular order, where constraints between the operation calls are non-obvious and state-specific.

In workflow and business process communities, a wide-spread approach to rollback for long-running transactions is *Sagas* [11], where system designers provide a *compensating action* for each operation (the term compensation here differs from its usage in dependability literature [3]). To undo the effects of a sequence of operations, the system executes the corresponding compensating actions in the reverse order. On cloud platforms, this is not always feasible. There are operations for which no compensating operation is provided in the API. Even when an operation seems like an inverse for another, there may be non-obvious constraints and side-effects, so that executing the apparently compensating operations in reverse chronological order would not restore the previous system state properly, and a different order, or even different operations, are more suitable [13].

Moreover, cloud API operations are often themselves error-prone: we have frequently observed failures or timeouts on most major commercial cloud platforms. The rollback therefore must handle failures that occur during the undo. This may require flexibility and executing alternative operations within the rollback.

To improve the dependability of cloud-based systems, we use an AI planner [27] to automate discovering an appropriate sequence of available operations from an API, in order to rollback to a checkpoint. Choosing a sequence of operations is a search in the space of possible solutions; highly optimized heuristics solve common cases of this computationally hard problem in reasonable time. A planner finds a sequence of calls while minimizing their number or cost, using knowledge of current and checkpointed states of the system, and a model of operations. Some variants of AI planning also allow for finding alternative sequences when failures occur during the rollback.

Our approach requires some changes to non-reversible forward operations such as deleting a disk volume. It also relies on a suitable abstract model of the domain, where each operation has a precise representation in its effects on each aspect of the abstract state. Rollback comes with little overhead, and restores the abstract state of resources, but not the financial charges. This work was inspired by our experience in developing tool support for users of cloud platforms (see `http://yuruware.com`). A technical report [28] contains details omitted here for space limitations.

## 2 Motivating Examples

We describe system operations in a cloud that illustrate difficulties of choosing an undo sequence for rollback.

**Example 1.** *The apparent compensator operation does not always reverse a forward operation. Auto-scaling* on AWS can create and maintain the number of virtual machines in a cluster at a target level, automatically supplying new machines to replace any that shut down. One might expect that "creating a cluster" could be compensated by "deleting a cluster". However, the deletion of a cluster can only be performed when there are no machines. Therefore, undoing the creation of a cluster requires a sequence of operations: first setting the target size of the cluster as zero, then, removing the cluster after waiting till all machines have been shut down due to the auto-scaling.

**Example 2.** *Executing compensator operations in reverse chronological order does not result in a sequence of operations being undone.* For instance, attaching a virtual disk volume to a virtual machine can be done safely any time. It is possible to invoke a "detaching a volume" operation any time, however, doing so could cause a serious failure such as disk inconsistency. Either the machine needs to be stopped or the volume needs to be properly unmounted before detaching it.

**Example 3.** *The best undo sequence is not a reverse sequence of compensators.* Assume the administrator of a system deployed in AWS develops a script to backup a disk volume from one geographical region to another. This script roughly includes the steps shown below, as well as some fault handling.

Listing 1: Remote backup script

```
1  Create a snapshot S0 of the source volume V0
2  Create a new volume V1 at the source region
       from the snapshot S0
3  Delete S0
4  Create a new empty volume V2 in the
       destination region
5  Launch VMs in source and destination regions
       (MS, MD)
6  Attach volume V1 to MS
7  Attach volume V2 to MD
8  Copy the data from V1 to V2
9  Detach volume V1 from MS
10 Detach volume V2 from MD
11 Delete V1
12 Terminate MS, MD
```

Taking an off-site backup in this way involves transferring large amounts of data between two geographically dispersed sites. To undo the effects of this script, it is wasteful to "copy back" the data before deleting it.

## 3 System Design

The main components of the system we propose concern (i) making forward operations (cloud API calls) reversible and (ii) automatically finding a sequence of operations for realizing rollback.

### 3.1 Overview of the Proposed System

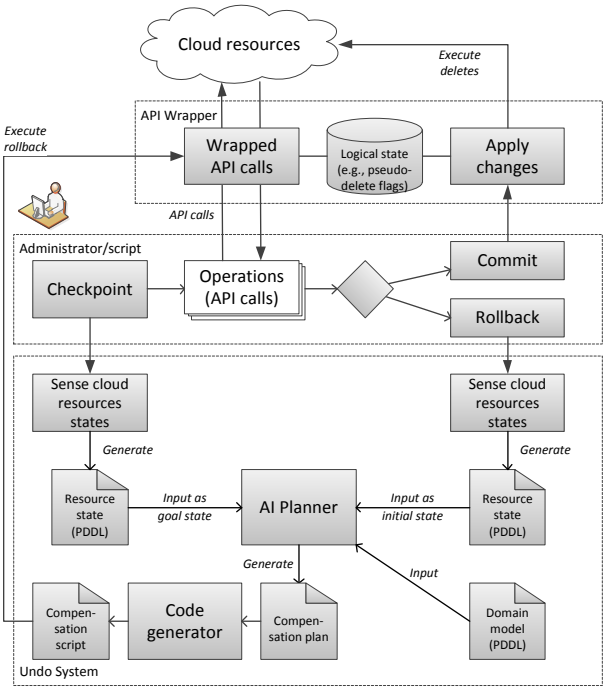


Figure 1: Overview of rollback via planning

Fig. 1 shows the overview of the proposed system which sits between the user or operational script, and the cloud management API . An administrator or an operational script first triggers a *checkpoint* to be taken[1]. Our undo system gathers relevant information, i.e., state of cloud resources and their relationship, at that time. After a checkpoint, the system transparently replaces all non-reversible operations, e.g., deleting a resource, with reversible ones (Section 3.2), and it offers rollback to the checkpoint. When a *commit* is issued, the non-reversible changes are applied to the cloud resources – thus, rollback is not offered anymore. When a *rollback* is issued, the system again gathers the state of cloud resources and feeds the pair of state information to an AI planner to construct an undo sequence of operations (Section 3.3).

[1]While the resources can form a vastly distributed system, their state information is obtained from a single source of truth: AWS's API.

### 3.2 Introduction of Pseudo-Delete

Many operations in cloud APIs have a dual: an operation that apparently does the opposite. However, deletion operations are not generally reversible, because creation would not revive the deleted resource in its state at the time of deletion, but instead it would create a fresh instance in an initial state. In order to make them reversible, we apply the *pseudo-delete* [12] technique.

The undo system offers a wrapper for cloud APIs. After a checkpoint, when the user asks to delete a resource, the wrapper sets a *delete flag*, indicating that the resource is logically deleted. Subsequent API calls to the resource are altered by the wrapper, e.g., by returning a "not found" error or filtering the resource out from a query result. When a delete operation needs to be reversed, the wrapper simply removes this flag; the user can request this by issuing a rollback. Only when a commit is issued, all resources with a delete flag are physically deleted.

### 3.3 Rollback via AI Planning

For rollback, the goal is to return to the state of the system when a checkpoint was issued. We propose to find the undo sequence via well-known AI planning techniques[27]. The usual *planning problem* is the following: given formal descriptions of the *initial state* of the world, the desired *goal state*, and a set of available *actions*, find a sequence of actions that leads from the initial to the goal state. In undo planning, the initial state is the state of the system when rollback was issued. The desired goal state is the system state captured in a checkpoint. The available actions are the API operations offered by a cloud provider. To minimize modelling overhead while achieving high performance, we follow state-based planning where a domain model (Section 4) shows each action annotated with a *precondition*, a logical formula describing when the action can be applied, and an *effect*, a logical formula specifying how applying this action changes the state of the world. The formulas refer to (finite-domain or Boolean) state variables.

In the undo system (Fig. 1), the planning problem (initial and goal states, set of available actions) is the input to the planner. Its output is a workflow in an abstract notation, stating which action to call, when and for which resources. The abstract workflow is forwarded to a code generator, which transforms it into executable code.

## 4 Domain Model and its Use

While the problem and system architecture are generic, for implementing a proof-of-concept prototype we chose AWS as the domain and the planning domain definition language (PDDL) [22] as the planning formalism.

One of the most critical aspects for applying AI planning is obtaining a suitable model of the domain [19]. For the purposes of this research, we designed a domain model manually, formulated in PDDL. This model has about 800 lines of code, and contains representations of 31 actions (see Table 1). Out of these, 21 actions are for managing AWS elastic compute cloud (EC2) resources, such as virtual machines (called *instances*) and disk volumes, and 6 actions for managing the AWS auto-scaling (AS) mechanisms. These actions have been selected due to their high frequency of usage by the developers in our group. The remaining 4 actions are for system maintenance, e.g., switching a server cluster to/from maintenance modes. Those actions are not specific to AWS but defined for generic system administration.

| **Resource type** | **Actions** |
|---|---|
| Virtual machine | launch, terminate, start, stop, change VM size |
| Disk volume | create, delete, create-from-snapshot, attach, detach |
| Disk snapshot | create, delete |
| Virtual IP address | allocate, release, associate, disassociate |
| Security group | create, delete |
| AS group | create, delete, change-sizes, change-launch-config |
| AS launch config | create, delete |
| Tag | create, delete |

Table 1: AWS actions captured in the domain model

**Case Study:** the PDDL definition of the action to delete a disk volume is shown in Listing 2. From this example, it can be seen that parameters are typed, predicates are expressed in prefix notation, and there are certain logical operators (*not, and, oneof,* ...), also in prefix notation. The precondition requires the volume to be in state *available* and not be subject to an *unrecoverable failure*. The effect is either an *unrecoverable failure* or the volume is in state *deleted* and not *available* any more.

Listing 2: Action to delete a disk volume in PDDL

```
(:action Delete-Volume
 :parameters (?vol - tVolume)
 :precondition
   (and
     (volumeAvailable ?vol)
     (not (unrecoverableFailure ?vol)))
 :effect
   (oneof
     (and
       (deleted ?vol)
       (not (volumeAvailable ?vol)))
     (unrecoverableFailure ?vol)))
```

*Unrecoverable failure* is a predicate we define to model the failure of an action. This indicates that the affected resource cannot be used for the remainder of the plan. It should be noted that our planning domain model resides on a slightly higher level than the AWS APIs. When a planning action is mapped to executable code, pre-defined error handlings are added as well. For example, a certain number of retries and clean-ups take place if necessary. Such a pre-defined error handler, however, works only on the resource in question. If it fails to address an error, one needs to find a backup action sequence that may take the state of other resources into account (see Section 5 for details.)

From the viewpoint of the planner the unrecoverable failure poses two challenges: *non-deterministic actions* and goal reachability. The outcome of *Delete-Volume* (success or unrecoverable failure) is observed as a non-deterministic event. In the presence of non-deterministic actions, the planner has to deal with all possible outcomes, which makes finding a solution harder than in the purely deterministic case. This requires a specific form of planning, called *planning under uncertainty*.

Further, the question "when is a plan a solution?" arises. To cater for actions with alternative outcomes, a plan may contain multiple branches – potentially including branches from where it is impossible to reach the goal. A branch that contains no unrecoverable failure is the normal case; other branches that still reach the goal are backup branches. A plan that contains more than one branch on which the goal can be reached is called a *contingency plan*. Branches from which the goal cannot be reached indicate situations that require human intervention – e.g., if a specific resource has to be in a specific state to reach the goal, but instead raises an unrecoverable failure, no backup plan is available.

In planning under uncertainty there are two standard characterisations of plans: a *strong plan* requires *all* branches of a plan to reach the goal, whereas a *weak plan* requires *at least one* branch to reach the goal. Standard planners that can deal with uncertainty are designed to find plans satisfying either of them; however, neither is suitable in our domain. It is highly likely that no strong plan can be found: many of the actions can return an unrecoverable failure, and many of possible branches cannot reach the goal. Weak plans have the disadvantage that only the "happy path" is found: a plan that allows reaching the goal only if nothing goes wrong. When finding a weak plan, a planner does not produce a contingency plan, which we deem insufficient.

In prior work of one of the authors [17], a different notion of a weak plan was introduced: the goal should be reached *whenever it is possible*. This is desired in the setting given here, as it will produce as many branches (i.e., a contingency plan) as possible that still reach the goal, given such alternative branches exist. For finding plans with these solution semantics, a highly efficient standard planner, called *FF* [16], was extended in [17].

There is one discrepancy between the standard AI planning and our domain, which requires attention. When new resources are created after a checkpoint, the resources exist in the initial state (i.e., the state captured when a rollback is issued) but not in the goal state (i.e., the state when a checkpoint is issued). Unless treated, the AI planner simply leaves these *excess resources* intact: since they are not included in the goal state, they are irrelevant to the planner. However, to undo all changes, excess resources should be deleted. To discover plans that achieve this, we perform one step of preprocessing before the actual planning: the goal state is compared with the initial state in order to find excess resources; the goal state is then amended to explicitly declare that these excess resources should end up in the "deleted" state.

## 5 Experiments

In this section, we give a preliminary evaluation of our approach. The system has been implemented as a prototype and has been rolled out for internal beta-testing. We used the prototype on numerous scenarios derived from situations encountered by our group. An example scenario from this set is presented. We also conducted performance exmperiments, which we discuss subsequently.

**Case Study:** rollback planning scenario. Assume a rollback is desired after Step 8 in Example 3 – i.e., the copying step. The *initial state* is the following: volumes $V_1, V_2$ are in use, $V_0$ exists but is not in use; snapshot $S_0$ is deleted; mover instances $M_S, M_D$ are running; $V_1$ is attached to $M_S$, $V_2$ is attached to $M_D$.

The *goal state*, to revert back to before Step 1 was executed, is therefore: $V_0$ exists but is not in use; $V_1, V_2, S_0$ are deleted; $M_S, M_D$ are stopped.

In this situation, the planner creates the plan below.

Listing 3: Undo plan after Step 8 in Example 3

```
1 Stop-Instance M_S
2 Stop-instance M_D
3 Detach-volume V_1 from M_S - fail:4; success:5.
4 Force-detach-volume V_1 from M_S
5 Delete-volume V_1
6 Detach-volume V_2 from M_D - fail:7; success:8.
7 Force-detach-volume V_2 from M_D
8 Delete-volume V_2
```

**Performance Experiments.** For testing the scalability of our solution, we explored two dimensions: (i) length of the solution (number of actions in the plan) and (ii) number of unrelated resources (described in [28]). We specified goals that would scale up the respective dimension. For each planning problem, we ran 10 tests and

took the average value. The planner ran inside a VirtualBox[2] VM under Ubuntu 11.10, with 4GB RAM available, and half of an Intel Core i5-2520M CPU at 2.5GHz.

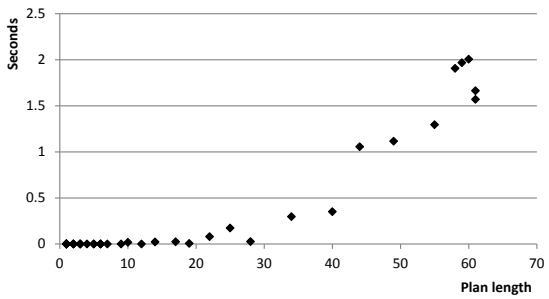


Figure 2: Scaling plan length – scatter plot.

The results of the plan length experiment are shown in Fig. 2. For small problems, the runtime of the planner was often close to or below the resolution of measurement (10 ms). Even plans with more than 60 actions were found in less than 2 seconds. To put this into perspective: scripts with over 20 steps are unusual in our group. Besides plan length, other factors impact planning complexity – resulting in the slightly scattered nature of the data points in Fig. 2. For detailed discussions of FF planning performance see, e.g., [16, 17].

While some administrators manage 1000s of machines, they usually employ higher-level primitives – e.g., in AWS, this would rather be done with auto-scaling groups than managing each machine individually. In general, AI planning is a PSPACE-hard problem – i.e., the runtime will increase exponentially when increasing the problem complexity. However, for plans of an arguably practical length, we found the runtime sufficient.

In comparison to the runtime of sensing and acting in AWS – for instance, the average runtime over 10 runs of the plan shown in Listing 3 was 145 seconds – the cost for planning seems marginal in many situations.

## 6 Related Work

Our dependability approach is a checkpoint-based rollback method [10, 26]. Alternative rollback methods are log-based [10, 12] or using "shadow pages" in databases [12]. In much research, checkpoints store a relevant part of the memory, and for rollback it suffices to copy the saved information back into the memory. In contrast, rollback in our setting means achieving that the "physical state" of a set of virtual resources matches the state stored in a checkpoint – i.e., achieving rollback requires executing operations, not only copying information. Thus, our setting is similar to long-running transactions in workflows or business processes, as discussed in Section 1.

Besides AI Planning, other techniques were used to achieve dependability in distributed systems management. [18] uses POMDPs (partially-observable markov decision processes) for autonomic fault recovery. An architecture-based approach to self-repair is presented in [4]. [23] is a self-healing system using case-based reasoning and learning. The focus in these works is self-repair or self-adaptation, not on undo for rollback.

AI Planning has been used several times for system configuration, e.g., [1, 2, 6, 7, 8, 9, 20], and for Cloud configuration, e.g., [14, 15, 21]. Some works aim at reaching a user-declared goal, e.g., [1, 6, 14, 15, 20], whereas others target failure recovery, e.g., [2, 7, 8, 21]. The goal in the latter case is to bring the system back into an operational state after some failure occurred.The closest related papers are [14, 15, 21]. In [15], planning is applied in a straight-forward fashion to the problem of reaching a user-specified goal. [21] applies hierarchical task network (HTN) planning on the PaaS level for fault recovery. [14] uses HTN planning to achieve configuration changes for systems modelled in the common information model (CIM) standard.

## 7 Conclusions

We presented an approach to rollback for cloud management, that wrappers the cloud management API, and uses AI Planning techniques to find an appropriate undo sequence. We formalized part of Amazon AWS APIs in a planning domain model, and used an off-the-shelf planner, in a prototype that creates undo sequences for rollback. This scales well as the number of operations needed to achieve the rollback increases.

In ongoing work, we broaden the application area of the planning approach, by finding robust "forward" plans to bring cloud resources into a desired state, modelled by an administrator in a user-friendly way. We also seek parallelisable plans. In the presence of non-deterministic outcomes of actions, this is a non-trivial problem.

Finally, we want to raise the point that rollback in our current system looks at resources "from the outside" – changes inside, e.g., data in a disk volume, are not considered as yet. While in principle the approach is agnostic to the amount of time passing between checkpointing and rollback, a longer duration makes it more likely that the internal state of resources changes in a way which would be inconsistent with the outside checkpointed state – the rollback might thus be less suitable after 20 hours than after 20 seconds. In future work, we consider creating full snapshots of the resources themselves when creating a checkpoint, so as to allow rolling back to their internal state as well.

[2] `https://www.virtualbox.org`

## Acknowledgements

NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program. This work is partically supported by the research grant provided by Amazon Web Service in Education [3].

## References

[1] ARSHAD, N., HEIMBIGNER, D., AND WOLF, A. L. Deployment and dynamic reconfiguration planning for distributed software systems. In *ICTAI'03: Proc. of the 15th IEEE Intl Conf. on Tools with Artificial Intelligence* (2003), IEEE Press, p. 3946.

[2] ARSHAD, N., HEIMBIGNER, D., AND WOLF, A. L. A planning based approach to failure recovery in distributed systems. In *WOSS'04: 1st ACM SIGSOFT Workshop on Self-Managed Systems* (2004), pp. 8–12.

[3] AVIZIENIS, A., LAPRIE, J.-C., RANDELL, B., AND LANDWEHR, C. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing 1*, 1 (2004), 11–33.

[4] BOYER, F., DE PALMA, N., GRUBER, O., SICARD, S., AND STEFANI, J.-B. A self-repair architecture for cluster systems. In *Architecting Dependable Systems VI*, vol. 5835 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2009, pp. 124–147.

[5] BROWN, A., AND PATTERSON, D. Rewind, repair, replay: three R's to dependability. In *10th ACM SIGOPS European workshop* (2002), pp. 70–77.

[6] COLES, A. J., COLES, A. I., AND GILMORE, S. Configuring service-oriented systems using PEPA and AI planning. In *Proceedings of the 8th Workshop on Process Algebra and Stochastically Timed Activities (PASTA 2009)* (August 2009).

[7] DA SILVA, C. E., AND DE LEMOS, R. A framework for automatic generation of processes for self-adaptive software systems. *Informatica 35* (2011), 3–13. Publisher: Slovenian Society Informatika.

[8] DALPIAZ, F., GIORGINI, P., AND MYLOPOULOS, J. Adaptive socio-technical systems: a requirements-driven approach. *Requirements Engineering* (2012). Springer, to appear.

[9] DRABBLE, B., DALTON, J., AND TATE, A. Repairing plans on-the-fly. In *Proc. of the NASA Workshop on Planning and Scheduling for Space* (1997).

[10] ELNOZAHY, E. N. M., ALVISI, L., WANG, Y.-M., AND JOHNSON, D. B. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv. 34*, 3 (Sept. 2002), 375–408.

[11] GARCIA-MOLINA, H., AND SALEM, K. Sagas. In *SIGMOD'87: Proc. Intl. Conf. on Management Of Data* (1987), ACM, pp. 249–259.

[12] GRAEFE, G. A survey of b-tree logging and recovery techniques. *ACM Trans. Database Syst. 37*, 1 (Mar. 2012), 1:1–1:35.

[13] GREENFIELD, P., FEKETE, A., JANG, J., AND KUO, D. Compensation is not enough. *Enterprise Distributed Object Computing Conference, IEEE International 0* (2003), 232.

[14] HAGEN, S., AND KEMPER, A. Model-based planning for state-related changes to infrastructure and software as a service instances in large data centers. In *Proceedings of the 2010 IEEE 3rd International Conference on Cloud Computing* (Washington, DC, USA, 2010), CLOUD '10, IEEE Computer Society, pp. 11–18.

[15] HERRY, H., ANDERSON, P., AND WICKLER, G. Automated planning for configuration changes. In *LISA'11: Large Installation System Administration Conference* (2011).

[16] HOFFMANN, J., AND NEBEL, B. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research (JAIR) 14* (2001), 253–302.

[17] HOFFMANN, J., WEBER, I., AND KRAFT, F. M. SAP speaks PDDL: Exploiting a software-engineering model for planning in business process management. *Journal of Artificial Intelligence Research (JAIR) 44* (2012), 587–632.

[18] JOSHI, K. R., SANDERS, W. H., HILTUNEN, M. A., AND SCHLICHTING, R. D. Automatic model-driven recovery in distributed systems. In *Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems* (Washington, DC, USA, 2005), SRDS '05, IEEE Computer Society, pp. 25–38.

[19] KAMBHAMPATI, S. Model-lite planning for the web age masses: The challenges of planning with incomplete and evolving domain models. In *AAAI'07: 22nd Conference on Artificial Intelligence* (2007).

[20] LEVANTI, K., AND RANGANATHAN, A. Planning-based configuration and management of distributed systems. In *Proceedings of the 11th IFIP/IEEE international conference on Symposium on Integrated Network Management* (Piscataway, NJ, USA, 2009), IM'09, IEEE Press, pp. 65–72.

[21] LIU, F., DANCIU, V., AND KERESTEY, P. A framework for automated fault recovery planning in large-scale virtualized infrastructures. In *MACE 2010, LNCS 6473* (2010), pp. 113–123.

[22] MCDERMOTT, D., ET AL. *The PDDL Planning Domain Definition Language*. The AIPS-98 Planning Competition Comitee, 1998.

[23] MONTANI, S., AND ANGLANO, C. Achieving self-healing in service delivery software systems by means of case-based reasoning. *Applied Intelligence 28* (2008), 139–152.

[24] OPPENHEIMER, D., GANAPATHI, A., AND PATTERSON, D. A. Why do internet services fail, and what can be done about it? In *Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems - Volume 4* (Berkeley, CA, USA, 2003), USITS'03, USENIX Association, pp. 1–1.

[25] PATTERSON, D., AND BROWN, A. Embracing failure: A case for recovery-oriented computing (ROC). In *HPTS'01: High Performance Transaction Processing Symposium* (2001).

[26] RANDELL, B. System structure for software fault tolerance. *IEEE Transactions On Software Engineering 1*, 2 (1975).

[27] TRAVERSO, P., GHALLAB, M., AND NAU, D., Eds. *Automated Planning: Theory and Practice*. Morgan Kaufmann, 2005.

[28] WEBER, I., WADA, H., FEKETE, A., LIU, A., AND BASS, L. Towards automatic undo for cloud management via AI planning. Tech. Rep. UNSW-CSE-TR-201226, School of Computer Science and Engineering, University of New South Wales, Sydney, NSW 2052, Australia, Sept. 2012.

[3] `http://aws.amazon.com/grants/`