Rollback Mechanisms for Cloud Management APIs using AI planning

Suhrid Satyal[1,2], Ingo Weber[1,2], Len Bass[3], and Min Fu[1,4]

[1]Data61, CSIRO, Sydney, Australia
[2]UNSW Australia, Kensington NSW 2052, Australia
[3]Carnegie Mellon University, Pittsburgh, PA, USA
[4]Macquarie University, NSW 2109, Australia

Abstract—Human-induced faults play a large role in systems reliability. In cloud platforms, system administrators may inadvertently make catastrophic mistakes, like deleting a virtual disk with important data. Providing rollback for cloud operations can reduce the severity and impact of such mistakes, by allowing to revert to a known, good state. However, in the context of cloud management this is non-trivial, since cloud consumers only have limited visibility and indirect control. In this paper, we present a scalable approach to rollback operations that change the state of a system on proprietary cloud platforms. In our previous work, we provided a system that augments cloud APIs and provides rollback operation using an AI planner. In this paper, we build upon our previous work, but parallelize the rollback plan generation based on characteristics unique to rollback scenario. Furthermore, we introduce a distributed anytime algorithm that gradually improves plan quality over time, until either an optimal plan is found or a timeout is reached. Through experimental evaluation we show that our approach scales better than a naïve approach, and effectively avoids the exponential behavior of AI planning. Further, we explore the trade-offs between the quality of rollback plans and plan generation time.

Index Terms—reliability, AI planning, cloud computing, web service, system administration

1 Introduction

For cloud applications, especially at large scale, one of the primary impediments to system dependability is human error. Human operator error is attributed with being the root cause of 20-50% of system outages [1], [2]. In many cases of outages caused by human operators, they may not be able to respond quickly enough to minimize the impact (e.g. losing traffic, violating SLAs, etc). Moreover, operators may perform an action that cannot be fixed entirely, like irreversibly deleting a data store. Such results can be caused with a single API call on cloud platforms like the Amazon Web Services (AWS) Elastic Compute Cloud (EC2) platform [3].

In the context of API-controlled cloud platforms, operators risk creating outages by executing undesired actions or actions whose effects are not fully known to them. Once an undesired state of the cloud resources has been reached, it is not necessarily clear to the operator how to revert to an earlier, good state, for the following reasons. APIs from public cloud providers are provided *as-is*, and correcting mistakes is only possible by calling the right API operations in the right sequence. In some cases, API operations are irreversible, and no combination of actions can completely recover from the mistake (e.g. deleting a virtual disk).

One way of mitigating human errors is by (re-)designing the platform in a way that allows administrators to rollback their changes. However, in an API-controlled proprietary system, this is not feasible: the operator does not have a level of control over the platform that allows for such changes. Another approach is to create a client-side system that augments the API and provides rollback facilities. An example of such an approach is our previous work on automatic undo for cloud management [4]. This approach works well for small to medium number of consecutive cloud operations (e.g. less than 50). However it does not scale well when the number of operations increases further. The problem in such cases is that the underlying automated planning method from Artificial Intelligence (AI) faces a computationally hard problem [5].

The key challenge that we address is that generating a rollback plan with a large number of steps becomes inefficient at some point. This is due to the inherent complexity of the AI planning problem: AI Planning is known to be PSPACE-complete even in its simplest form [5]. That means that there is no efficient planning algorithm in the general case (unless **P=NP**). The implication for our tool is that, as the number of cloud operations that change the state increases, rollback plan generation time increases exponentially, as confirmed by our experiments.

One approach to facilitate scalable rollback on an API-controlled cloud platform is to use *intermediate checkpoints*. After an initial, user-triggered checkpoint is created, calls to the cloud API can be intercepted by the system, thus creating intermediate checkpoints as appropriate. Should the need for rollback arise, the system can create a number of AI planning tasks based on the intermediate checkpoints, use an AI planner to discover appropriate sequences of recovery actions for each planning task, and aggregate them into a complete *rollback plan*. Such rollback plans can be

found quickly. However, a plan may comprise *several times as many steps as the optimal solution*. Therefore, we propose an approach that achieves a tradeoff between planning time and the quality of the rollback plan through a *distributed anytime algorithm*.

In this paper, we make the following contributions:

1) We design a system that utilizes a unique feature of rollback scenarios, i.e., the possibility to generate *intermediate checkpoints*, to solve efficiency challenges of AI planning for undo.
2) We propose strategies for dividing and parallelizing rollback plan generation: based on the intermediate checkpoints, we divide the planning task into smaller, independent tasks, parallelize planning process, and assemble the resulting partial plans into a comprehensive rollback plan.
3) We demonstrate the need for a distributed anytime algorithm and propose strategies for scheduling and distributing rollback plan generation in a cluster. We represent checkpoints as vertices and partial rollback plans as edges of a directed acyclic graph. Then, we propose algorithms that update this graph with new edges and improve rollback plan quality within a given timeout.
4) We evaluate rollback mechanisms and:
   a) show that we can effectively avoid the exponential behavior of naïve planning,
   b) show that our approach is highly parallelizable by comparing plan generation time on machines with different numbers of CPU cores, and
   c) show that, using our distributed anytime algorithm, we can facilitate a trade-off between rollback plan quality and planning time.

In an earlier version of this work [6], we addressed the scalability issues of rollback planning and evaluated the approach (contributions 1, 2, 4a and 4b above). In this paper, we introduce a distributed anytime algorithm that improves the quality of rollback plans until a specified timeout is reached, and finally executes the best plan that was generated within the given time. These new contributions (3 and 4c above) comprise about 60% of this paper.

The remainder of this paper is organized as follows. The next section discusses scenarios and challenges that motivate our research. We discuss the undo system and different types of checkpoints in Section 3. In Section 4, we explain our rollback strategies. In Section 5, we analyze the performance of our rollback strategies and compare them with the naïve approach. In Section 6, we explain the results, limitations, and validity of our approach. We discuss related work in Section 7, and finally draw conclusions in Section 8.

2 Background & Motivating Examples

In this section, we discuss some scenarios and challenges of generating undo operations, in part based on our previous work [7]. We also discuss the how previous work has addressed these challenges.

1) *Attaching or Detaching Virtual Disks*
   It is possible to invoke a detach or attach operation any time, however doing so could cause failures such as disk inconsistency. A rollback plan needs to either stop the virtual machine, or unmount the volume properly before detach or attach operations. While cloud platforms offer logging service (e.g. AWS CloudTrail [8]), using a log-based approach to execute compensatory operations in reverse order will not suffice in such scenarios.
   Insight: Executing compensatory operations in reverse chronological order does not rollback some operations.

2) *Creating and Deleting Clusters*
   Auto-scaling mechanisms, like on AWS, can create and maintain a number of virtual machines in a cluster, and automatically supply new machines to replace ones that shut down or fail. In platforms like AWS, creation of cluster cannot be compensated by deletion. Deletion can only be performed when a cluster has no machines. A rollback plan that needs to delete a cluster has to first set the cluster size to zero, and execute delete operation only after all machines have been shut down.
   Insight: Compensatory operation (provided by the API) does not always reverse a forward operation.

3) *Creating and Deleting Backups*
   Creating backup virtual machines (VMs) requires a sequence of operations such as creating snapshot, creating empty instances and volumes, and copying data. To undo the effects, it suffices to simply delete the backup. Execution of compensating action in reverse order entails executing unnecessary operations (e.g. copying data back).
   Insight: The best rollback plan is not necessarily a reverse sequence of compensatory actions.

4) *Deleting a Virtual Disk*
   For delete operations, no compensating action may be provided by the API. Although it is possible to recover data stored in virtual disk by using backups, the disk itself cannot be restored once it is deleted. Also, any data added to or changed on the disk after creating a backup is lost irreversibly.
   Insight: Some operations provided by APIs are irreversible.

5) *Deploying Application at Scale*
   Deployment at scale can be complex and require a large number of API invocations. For example, deployment of new version of a web application may require creating virtual instances and volumes, creating backups, changing auto-scaling configuration, scaling instances up or down, configuring load balancers, changing DNS entries, etc. In such scenarios, the probability of errors is high – especially if the deployment is not automated.
   Insight: Large-scale deployments require a large number of API invocations, and have higher probability of failure.

Utilizing an AI planner, we can generate a sequence of rollback actions. Typically, the planner generates the shortest possible plan, i.e., with the least number of actions. This requires a domain model of the API. Analyzing such a model can provide guarantees which actions can be undone,

and under which circumstances [7].

Irreversible operations can be facilitated by providing a wrapper around irreversible API operations, where the wrapper replaces the irreversible actions with pseudo variants, e.g., delete is replaced with *pseudo-delete*. In [4], we proposed an approach that wraps AWS's cloud management API and uses a formal domain model to generate undo sequences for rollback. A single AI planning process generates undo sequence based on the domain model and information of the current and a previously checkpointed state of resources. Delete operations are replaced with pseudo-delete: calling a delete operation will only flag the resource as deleted; from that point on, wrapped actions affecting the resource behave as if it actually was deleted. When an administrator requests rollback, the resource is undeleted; in contrast, only when the administrator commits the changes, all flagged resources are actually deleted.

In large scale deployment scenarios, the approach fails to scale plan generation time with number of operations. In the earlier version of this work [6], we addressed this issue by introducing *intermediate checkpoints*. Using these checkpoints, we divide and parallelize the planning processes, and aggregate the results. However, in the earlier version we did not consider the quality of rollback plans.

3 Undo System

In this section, we explain how the undo system discovers and executes a sequence of undo operations, based on the state of cloud resources.

3.1 Overview and API Wrapper

The system has two main parts: an API wrapper, which intercepts calls to Amazon EC2 API and offers few additional commands, and an AI planner, which generates rollback actions based on checkpointed states.

In a typical usage scenario, an administrator (or script) issues a *checkpoint command* before making changes to resources. The system then gathers relevant information about the state of the cloud resources, and stores this as a checkpoint. After creating checkpoints, administrators can make changes to resources. At this point, the API wrapper transparently replaces irreversible operations with reversible ones and offers an additional rollback operation. If an inadvertent change has been made, the administrator issues a *rollback command*. In contrast, if the changes are satisfactory, the administrator issues a *commit command* so that irreversible changes are applied to cloud resources. After the commit command is executed, the checkpoint is deleted and rollback is not offered anymore.

To achieve this functionality, the API wrapper intercepts EC2 commands, selectively delegates them to the API, and handles additional commands for checkpointing, rollback, commit, and undelete – see details below. When a rollback command is issued, it creates a planning problem file, and invokes the AI planner to generate list of rollback actions. The API wrapper collects and translates these actions into a bash script, and executes it.

The API wrapper also facilitates *pseudo-delete* operation. Delete operations on the EC2 API are not reversible. When a delete command is executed, the system sets a *delete flag*, indicating that a resource is deleted. The API wrapper intercepts subsequent calls to deleted resource and indicates that the resource does not exist. When an *undelete* command is issued, the system removes this flag.

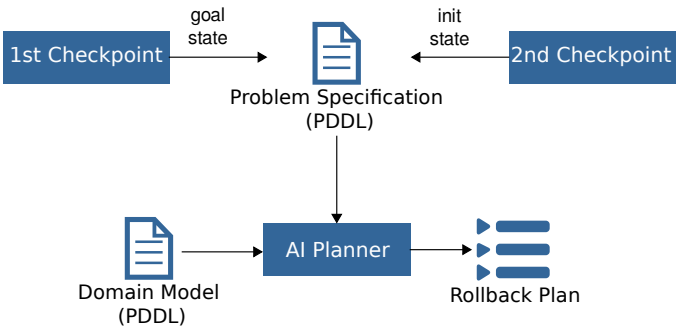


Fig. 1: Using an AI planner to find a rollback plan. The AI planner generates a list of rollback actions based on a problem specification and a domain model file. The problem specification is created using two checkpoints: the earlier checkpoint comprises the goal state, and the checkpoint created later comprises the initial state. The domain model is created manually by modeling AWS EC2 API.

3.2 AI Planner

We model rollback as an AI planning task and use a recent variant [9] of the FF planner [10] to find rollback actions. The FF planner solves planning problem efficiently by combining hill-climbing with systematic forward search, and using heuristics to prune the search space. Given a planning task specified in the planning domain definition language (PDDL) [11], it generates a list of actions that solves the task.

A *planning task* comprises objects, predicates, initial state, goal specification, and a set of possible actions. A planner like FF then finds a sequence from the set of actions that, starting from the initial state, can reach the goal state. The variant of FF that we use considers all paths on which the goal can be reached, even if an action does not yield the most desired outcome [9]. Figure 1 shows an illustration of the AI planner.

Objects, initial state, and goal specification are expressed in a PDDL problem file. The undo system creates a PDDL problem file based on checkpoints whenever a rollback operation is necessary. Given two checkpoints X and Y, created in this order,

1) Objects are all known cloud resources,
2) Initial state is the state of cloud resources specified in Y, the current state, and
3) Goal state is the state of cloud resources specified in X, the original checkpoint.

The PDDL problem file is generated by the undo system to achieve rollback, when this operation is called. Since the aim is to undo changes, the earlier checkpoint X is the goal, whereas the later checkpoint with the current state of resources is the starting point, the initial state. By reaching the goal, we can effectively revert the changes that have taken place since the original checkpoint was taken. Listing 1 shows an example of a PDDL problem file.

```
(define (problem EC2-0)
  (:domain EC2)
  (:objects
    inst00 - tInstance
    vol00 - tVolume
    devName00 - tDeviceName
    AZ0 - tAvailabilityZone)
  (:init
    (instanceRunning inst00)
    (volumeInUse vol00)
    (inAZ vol00 AZ0)
    (inAZ inst00 AZ0)
    (deviceNameInUse devName00 inst00)
    (volumeAttachedToInstance vol00 inst00 devName00
        ))
  (:goal
    (and
    (volumeAttachedToInstance vol00 inst01 devName00
        )
    (instanceRunning inst00))))
```

Listing 1: Sample PDDL problem file

```
(define (domain EC2)
  (:types tAMI tInstance ...)
  (:predicates
    (hasAMI ?x0 - tInstance ?x1 - tAMI) ... )
  (:constants ...)
  (:action Run-Instance
    :parameters (?ami - tAMI ?inst - tInstance ?
        secGroup - tSecurityGroup)
    :precondition
      (and
        (notYetCreated ?inst)
        (activeSecurityGroup ?secGroup))
    :effect
      (oneof
        (and (instanceRunning ?inst)
          (not (instanceStopped ?inst))
          (not (notYetCreated ?inst))
          (belongsToSecurityGroup ?inst ?secGroup)
          (hasAMI ?inst ?ami))
        (unrecoverableError ?inst))))
```

Listing 2: Snippet of PDDL domain file

Predicates and Actions are expressed in a PDDL domain model file. In previous work [4], we have manually created a PDDL domain file by modeling AWS EC2 APIs and restricting the domain model to provide guarantees that actions are undoable. Listing 2 shows a snippet of our PDDL domain file. If a formal model of the API or the system behind it is available, such PDDL domain files may be generated automatically. In [9], this was done for over 2,000 services offered by SAP systems, using software engineering models from the development of those systems.

In the variant of PDDL we employ, new resources cannot be created out of nowhere. In the domain model, we use the unary predicate *notYetCreated* to enable creation of resources. Additionally, the unary predicate *deleted* is used to indicate the deletion of a resource. In relation to that, some minor post-processing of initial and goal states is necessary: objects that were *notYetCreated* in the original checkpoint, but have been created by the time rollback is called, need to be marked as *deleted* in the goal state since they cannot be "un-created". While these two states (not yet created and deleted) are very similar – the respective object is not there – it is important to distinguish them in our models: once a resource with a specific ID has actually been deleted, for most resource types on AWS it is impossible to get exactly this resource back. For resource types like static IP addresses, this distinction can be very important.

3.3 Checkpoints

A checkpoint is a reference that identifies state of resources on the cloud at a point in time. It captures the state information of all cloud resources, such as which VMs existed, which volumes existed, which VMs were connected to which volumes, etc.

When administrators want to have the ability to rollback, they create a checkpoint before making changes to resources on the cloud. The undo system then gathers information about state and relationships of cloud resources, and saves it to persistent storage. Additionally, the system creates other types of checkpoints depending on the number and type of commands that are called between taking a checkpoint and calling commit or rollback. If called, the rollback operation uses these checkpoints to generate a rollback plan, which is translated into an executable script.

The system uses three types of checkpoints: (i) manual checkpoint, (ii) intermediate checkpoint, and (iii) current checkpoint. All three types of checkpoints contain the same kind of information about cloud resources and their states, for use by the AI planner when generating rollback plans. Figure 2 shows checkpoint placement and the rollback mechanisms that they facilitate.

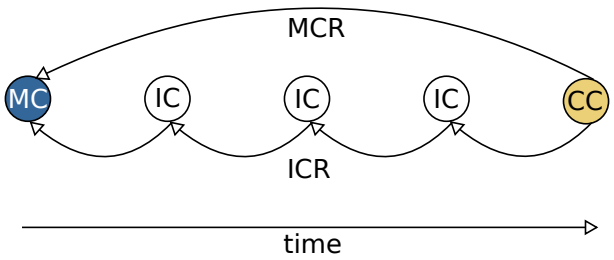


Fig. 2: Placement of Manual Checkpoint (MC), Current Checkpoint (CC), and Intermediate Checkpoint (IC). Manual Checkpoint Rollback (MCR) performs rollback directly from CC to MC. Intermediate Checkpoint Rollback (ICR) performs rollback via the ICs.

3.3.1 *Manual Checkpoint*

a manual checkpoint represents a consistent state to which a system can be rolled back. This is a checkpoint created by the system administrator manually before making changes to resources on cloud. The manual checkpoint information is stored persistently.

3.3.2 *Intermediate Checkpoint*

This is a checkpoint created by the undo system after a certain number of *change commands*, i.e., commands which change state of the system, have been executed. Intermediate checkpoints are stored persistently. They are used to improve rollback plan generation time.

3.3.3 Current Checkpoint

This is a temporary checkpoint object created by the system when the rollback command is triggered. The current checkpoint does not need to be stored persistently, as it is used only when the rollback command is executed.

Intermediate checkpoints plays a significant role in improving the scalability of rollback plan generation. They allow us to divide the planning process into smaller independent tasks, where each task is responsible for generating rollback actions between two adjacent checkpoints. Overall plan generation time can be reduced significantly when these tasks are parallelized.

The API wrapper tracks the number of called change commands, and creates an intermediate checkpoint after every *n-th* change command. The number of commands, n, after which an intermediate checkpoint is created is configurable. All details about the creation of intermediate checkpoints are hidden from the user.

One of the challenges in creating the checkpoints is that, when cloud system is very large and there are a lot of cloud resources to be captured, checkpoint creation time is relatively long. Therefore, choosing the number of change commands for intermediate checkpoints, n, represents a trade-off between user experience and plan generation time.

4 Rollback Strategies

Rollback is the core function of our Undo System. The rollback command returns the resources to their state at the point in time when a *manual checkpoint* was created. The undo system uses the AI planner to find a rollback plan of undo actions, generates a bash script based on the rollback plan, and executes the script.

Our approach comprises four strategies to perform rollback, which we describe below. They differ in how they use the various checkpoints to generate a rollback plan. In Section 5, we comparatively evaluate the approaches and other aspects.

4.1 Manual Checkpoint Rollback (MCR)

This rollback strategy is a naïve approach where the rollback command creates a PDDL problem file from solely the *current checkpoint* and the *manual checkpoint*. The state captured in the current checkpoint is used as initial state, and the state captured in the manual checkpoint is used as goal state. Rollback plans generated by this approach are typically optimal in terms of plan length.

The drawback of this approach is that it does not consider the relationship between computation time and plan length (number of actions in the undo plan). Because planning is PSPACE-hard, the runtime of undo plan generation using this approach will increase exponentially over increasing plan length. The MCR strategy is the base case from our prior work [4], and the goal of the other two strategies is to improve over MCR when creating long rollback plans. To improve plan generation time for large plans, the other two strategies discussed next make use of *intermediate checkpoints*.

4.2 Intermediate Checkpoint Rollback (ICR)

In this rollback strategy, user commands are tracked to create *intermediate checkpoints*, as described in Section 3.3. As stated earlier, this strategy is unique to the rollback scenario: only because we can observe the "do"-actions and can create intermediate checkpoints, we can make use of them for efficient rollback.

When the rollback command is invoked for ICR, the undo system reads each pair of adjacent checkpoints in reverse chronological order of their creation, and generates appropriate problem PDDL files. This set of PDDL problem files is used by the planner to generate a set of sequences of rollback actions.

For example, say there are 3 intermediate checkpoints C_1, C_2, and C_3, as well as the current checkpoint, CC, and the manual checkpoint, MC. Then ICR creates four checkpoint pairs: $(CC, C_3), (C_3, C_2), (C_2, C_1)$, and (C_1, MC). For every checkpoint pair (X, Y), the system generates a planning task with X as initial state and Y as goal state.

Rollback plans for each checkpoint pair are generated in parallel. Once all partial rollback plans are available, they are concatenated to form a single sequence of commands, the end-to-end rollback plan, which is then translated and executed as before.

4.3 Scalable Rollback (SR)

Scalable Rollback is a rollback strategy that combines MCR and ICR. Figure 3 shows how these strategies work, and how SR combines the other two approaches.

SR runs MCR and ICR in parallel when the rollback command is called. ICR, in turn, parallelizes its planning process into the generation of partial rollback scripts, and concatenates them to a single script. The system executes whichever rollback script is generated earliest. All parallel computation is terminated once either MCR or ICR has generated a rollback script. As such, SR performs a race between MCR and ICR.

One of the limitations of SR is that the rollback plan does not aim to optimize plan length. An optimal plan is typically achieved through MCR, but ICR typically returns sub-optimal plans. The ICR and SR strategies were introduced in the earlier short version of this paper [6], and the goal of the following strategy is to address their limitations by improving plan quality through an anytime algorithm.

4.4 Anytime Rollback (AR)

An important quality of rollback plans is plan length, where shorter length indicates higher quality. *Optimality* can thus be defined as the absence of a shorter plan that solves the same planning problem. In some rollback scenarios, generating optimal solutions may be infeasible within given time constraints. We must then be satisfied with the shortest plan that can be generated within the specified time.

Following the paradigm of anytime algorithms [12], [13], an initial plan is generated as fast as possible; subsequently, plan quality is progressively improved until the acceptable planning time has elapsed. Anytime algorithms allow the computation to be terminated and return a result whose quality is a function of time. Termination can be achieved through interrupts or a pre-defined time limit (or contract).

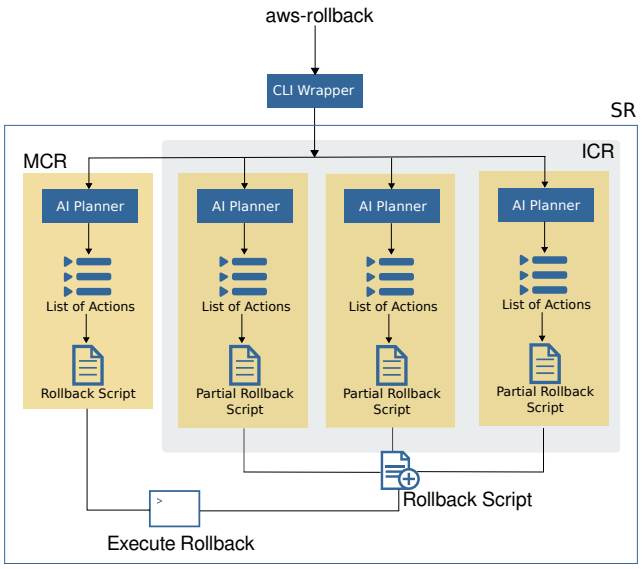


Fig. 3: Overview of three rollback strategies: MCR, ICR and SR. is issued, the API Wrapper creates a problem specification and invokes the planner for only MCR, or in several times in parallel for ICR, or both for SR. Partial rollback scripts generated by ICR are concatenated into a single script. For SR, once a rollback plan is generated by either ICR or MCR, the respective other planning processes are stopped and the rollback plan is executed.

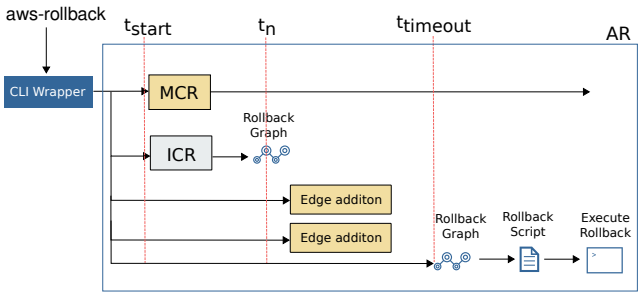


Fig. 4: Overview of AR. When ICR runs to completion, it creates a Rollback Graph. Then, the wrapper schedules planning tasks to improve plan until a timeout is reached. After the timeout is reached, the shortest path between CC to MC is chosen, converted to rollback script, and executed.

Anytime Rollback (AR) is an anytime contract algorithm where administrators specify a timeout before initiating a rollback operation. This timeout represents the maximum deliberation time during which planning is performed.

In AR, we represent checkpoints as vertices of a directed acyclic graph (DAG). Vertices are connected by edges that represent known *partial* rollback plans. To start, AR computes ICR so that some path from MC to CC is available. AR then adds new edges to the graph by generating plans between checkpoints that are not directly connected by an edge. When the timeout is reached, AR returns the shortest path available, if any. If MCR succeeds before the timeout, AR returns the plan immediately because this plan is typically optimal. Figure 4 shows how AR works.

We next describe our distributed anytime algorithm and the strategies we use to schedule the computation of rollback plans for specific edges in the DAG.

	Planning time	
#cores	$\#p = 0.5\times$ #cores	$\#p =$ #cores
4	69.22s	221.945s
8	224.34s	405.85s
16	424.725s	629.115s

TABLE 1: Average Time taken by AI planning tasks with a plan length of 256 steps in Blocksworld domain. $\#p$: number of planning processes; #cores : number of logical cores. When the number of planning processes is increased in linearly to the number of cores, average planning time increases.

4.4.1 Distributed Anytime Rollback

When multiple planning processes of similar problem complexity are executed in parallel on a single machine, all of them slow down – even on a many-core machine. To demonstrate this behavior, we designed a planning problem in the Blocksworld domain [14], such that an optimal plan is 256 steps long. In AI planning, Blocksworld is a well-known domain where objects are represented using stacked blocks, and actions are operations that move these blocks between stacks. Using the problem definition we created, we ran multiple planning processes in parallel. The results are shown in Table 1. For every machine, we measured performance such that the number of planning threads ($\#p$) is equal to the number of *logical* cores. To ensure that slowing down was not caused by hyper-threading, we disabled hyper-threading at the OS level and measured performance such that number of threads is equal to the number of *physical* cores, i.e., half the number of logical cores.

We can observe that the planner slows down significantly if we increase the number of threads in proportion to the number of available cores. Since AR can run large number of plans with varying lengths in parallel, a multi-threaded approach on a single large machine does not scale. These results were derived using the FF planner [10], which we use in our evaluations. Besides FF, we tested FD [15] with various configurations and heuristics, and the results were consistent with the ones shown in Table 1. Although the problem of computing edges in AR *should* be embarrassingly parallel, the practicalities of parallel planning posed a significant obstacle to us.

Based on these observations, we designed a *distributed AR* algorithm where *a single master node and multiple worker nodes*, each running on a distinct machine, work together to run planning tasks and aggregate results. The master node determines in which order to explore edges, assigns the respective planning tasks to worker nodes, and aggregates results as they are returned by the workers. Each worker node runs one planning task at a time, to avoid the performance degradation observed above. Distributed AR thus ensures that despite poor parallel performance of planners, the rollback mechanism itself can scale and improve plan quality – but it does introduce the additional complexity and overhead of distributed computing.

4.4.2 Edge Exploration Algorithms

The master node is responsible for distributing planning tasks to workers. Initially it distributes the intermediate

Algorithm 1: Depth First edge exploration

```
1 Fn depth_first()
2     skip_level ← 1;
3     max_feasible_hops ← number of checkpoints - 3 ;
4     while skip_level ≤ max_feasible_hops do
5         foreach Vertex v in Rollback Graph do
6             candidate_vertex ← vertex at skip_level^th
                hop to the left ;
7             add edge from v to candidate_vertex;
```

Algorithm 2: Breadth First edge exploration

```
1 Fn breadth_first()
2     skip_level ← 1;
3     max_feasible_hops ← number of checkpoints - 3 ;
4     foreach Vertex v in Rollback Graph do
5         add vertices within max_feasible_hops from v
            to candidate_vertices;
6         foreach Vertex candidate_vertex in
            candidate_vertices do
7             add edge from v to candidate_vertex;
```

planning tasks for ICR, such that a first overall plan $(MC \rightarrow IC_0, IC_0 \rightarrow IC_1, ..., IC_{n-1} \rightarrow IC_n, IC_n \rightarrow CC)$ is obtained quickly. Subsequently, the master node can use various strategies to add more edges to the rollback graph and distribute tasks to workers. For this paper, we have devised the following five edge exploration algorithms:

Depth First. This algorithm iteratively *skips* checkpoints by a factor and adds all possible edges in that skip level. The pseudo-code is described in Algorithm 1.

Breadth First. This algorithm iteratively adds all possible edges from one vertex to other eligible vertices. The pseudo-code is described in Algorithm 2.

Divide and Conquer. This algorithm picks a split point in the middle of the graph and adds edges between start and end vertices of each sub-graph. It then moves the split point to the right and left and repeats the edge addition process. The pseudo-code is described in Algorithm 3.

Subgraph Optimization. The Divide and Conquer algorithm does not optimize plans in each sub-graph that it creates after a split. Therefore, some edges are never explored. Subgraph Optimization explores these ignored edges by recursively improving every sub-graph that is created by Divide and Conquer. The pseudo-code is described in Algorithm 4.

Random. Serving as a baseline, this algorithm randomly selects which edges to explore. The pseudo-code is shown in Algorithm 5.

These algorithms are implemented using the actor model of the Akka framework[1]. In every algorithm, we make sure that each edge is explored at most once. That is, all planning processes are unique. It should be noted that our framework is pluggable, allowing for easy addition of new algorithms. The five algorithms presented here should thus be seen as a starting point, rather than the ultimate solution. In the next section, we compare these algorithms in different scenarios and demonstrate that some algorithms consistently outperform others.

1. http://akka.io/, last accessed on 8/8/2016.

Algorithm 3: Divide and Conquer edge exploration

```
   Data: MC = manual checkpoint, CC = current
         checkpoint
1  Fn divide_and_conquer()
2      n ← size(sorted_checkpoints);
3      m ← n/2;
4      candidate_vertex ← m^th elem in
         sorted_checkpoints;
5      add edge from candidate_vertex to MC;
6      add edge from CC to candidate_vertex;
7      left ← m; right ← m;
8      while true do
9          if left ≤ 0 and right ≥ n − 1 then
10             return;
11         left ← left − 1; right ← right + 1;
12         if left > 0 then
13             add edge from left^th elem of
                 sorted_checkpoints to MC;
14             add edge from CC to left^th elem of
                 sorted_checkpoints;
15         if right < n − 1 then
16             add edge from right^th elem of
                 sorted_checkpoints to MC;
17             add edge from CC to right^th elem of
                 sorted_checkpoints;
```

5 Evaluation

In this section we describe the experiments we conducted to comparatively evaluate the different rollback strategies and discuss results. To compare the MCR, ICR, and SR strategies, we focus solely on plan generation times, not quality, when varying (i) plan length and (ii) the number of cores available for planning. In contrast, for AR we compare its different edge exploration strategies it in terms of solution quality over time. Since AR includes MCR and ICR, and goes beyond SR to improve plan quality, the comparison of AR with these strategies can be derived from the AR experiments as well.

5.1 Scalable Rollback

The test scenarios we defined for SR modify the configuration of a collection of instances and volumes by detaching and attaching volumes to different instances. In our experiments, the number of cloud resources forming the states is the same for all plan lengths, so as to keep this factor stable – otherwise it would impact the initialization phase of planning. We created scenarios where an optimal rollback plan has a plan length of $m \in \{16, 32, 48, 64, 80, 96, 112, 128, 144\}$. To ensure comparability of planning tasks for MCR, ICR, and SR, the scenarios were set up so that the rollback plans generated by all approaches had same number of steps.

In order to create *checkpoints as observation of real states of AWS resources*, we ran scripts that started with creating a manual checkpoint, executed the "do" actions on AWS

Algorithm 4: Subgraph Optimization edge exploration

```
Data: MC = manual checkpoint, CC = current checkpoint
Fn subgraph_optimization(sorted_checkpoints)
    n ← size(sorted_checkpoints);
    if n = 1 then
        return;
    else if n = 2 then
        add edge from 2nd elem to 1st elem in sorted_checkpoints;
    else if n = 3 then
        add edge from 3rd elem to 2nd elem in sorted_checkpoints;
        add edge from 3rd elem to 1st elem in sorted_checkpoints;
        add edge from 2nd elem to 1st elem in sorted_checkpoints;
    else
        m ← n/2;
        candidate_vertex ← m^th elem in sorted_checkpoints;
        add edge from candidate_vertex to MC;
        add edge from CC to candidate_vertex;
        subgraph_optimization( sorted_checkpoints[0:m] );
        subgraph_optimization( sorted_checkpoints[m+1:] );
        left ← m; right ← m;
        while true do
            if left≤0 and right≥ n - 1 then
                return;
            left ← left − 1; right ← right + 1;
            if left > 0 then
                add edge from left^th elem of path to MC;
                add edge from CC to left^th elem of sorted_checkpoints;
                subgraph_optimization( sorted_checkpoints[0:left] );
                subgraph_optimization( sorted_checkpoints[left+1:] );
            if right < n - 1 then
                add edge from right^th elem of path to MC;
                add edge from CC to right^th elem of sorted_checkpoints;
                subgraph_optimization( sorted_checkpoints[:right+1] );
                subgraph_optimization( sorted_checkpoints[right+1:] );
```

Algorithm 5: Random edge exploration

```
Fn random_edge()
    max_feasible_hops ← number of checkpoints - 3;
    task_list ← empty list;
    foreach Vertex v in Rollback Graph do
        candidate_vertices ← vertices within max_feasible_hops from v;
        foreach Vertex cv in candidate_vertices do
            put edge task from v to cv in task_list;
    shuffle task_list;
    foreach edge in task_list do
        add edge to rollback graph;
```

necessary to reach the state from which we wanted to roll back while the undo system collected intermediate checkpoints, and saved the final state as current checkpoint. We used these checkpoints in our experiments, without always executing the resulting rollback plans to save time and cost.

Intermediate checkpoints were created with $n = 20$, i.e., after every 20^{th} command that changes the state of the system. As stated earlier, the number of commands n after which an intermediate checkpoint is created presents a trade-off in user experience: on the one hand creating a checkpoint takes time, therefore doing so less often (larger n) is preferable. On the other hand, if n is too large, then the rollback cannot profit much from ICR or SR, and planning for rollback will take very long, so a smaller n is preferable. In our AWS-based scenarios, we found that $n = 20$ was a good compromise.

Using the checkpoint sets collected as per above, we ran plan generation time for each plan length and rollback strategy for 25 iterations. We measured the time for plan generation and report average values over the 25 iterations here. Since standard deviation and standard error of mean were low for the values from the 25 iterations, this was deemed sufficient.

For plan generation, we also used AWS EC2 machines, namely C4.xlarge instances with 7.5 GB memory and 4 vCPUs, C4.2xlarge instances with 15 GB memory and 8 vCPUs, and C4.4xlarge instances with 30 GB memory and 16 vCPUs. It should be noted that a vCPU is a hyperthread on a CPU core of an Intel Xeon processor.[2] Therefore, 4 vCPUs correspond to two physical cores, for instance. C4 machines offer more stability in terms of actual performance than standard machines, at a premium price.

5.1.1 Plan Generation Time vs. Rollback Strategy

Using the above setup, we compared rollback plan generation time for different plan lengths and the three strategies: Manual Checkpoint Rollback (MCR), Intermediate Checkpoint Rollback (ICR), and Scalable Rollback (SR). The results of this experiment are shown in Figure 5.

First we observe that the runtime of MCR indeed exhibits exponential behavior: the respective curve is roughly a straight line, but on a logarithmic scale. In contrast, the ICR and SR strategies scale up almost linearly. As plan length

2. http://aws.amazon.com/ec2/instance-types/, accessed on 14/6/2015.

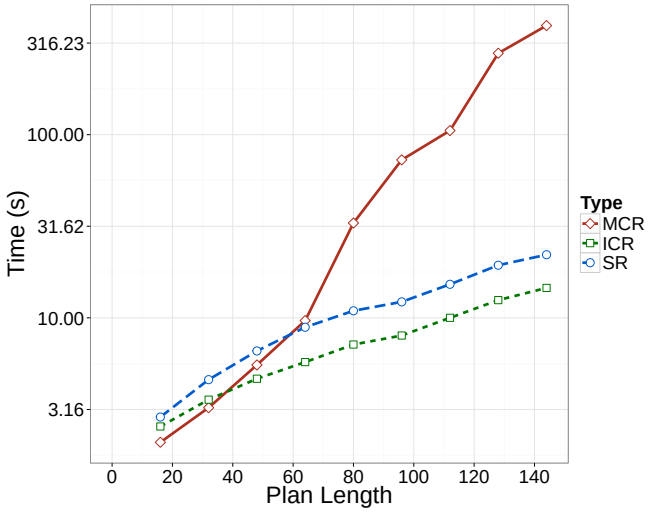


Fig. 5: Runtime of plan generation (in seconds) using the strategies MCR, ICR, and SR. Note that the y-axis uses a *logarithmic scale*. Plan generation time was measured on a VM with 4 virtual CPU cores (C4.xlarge).

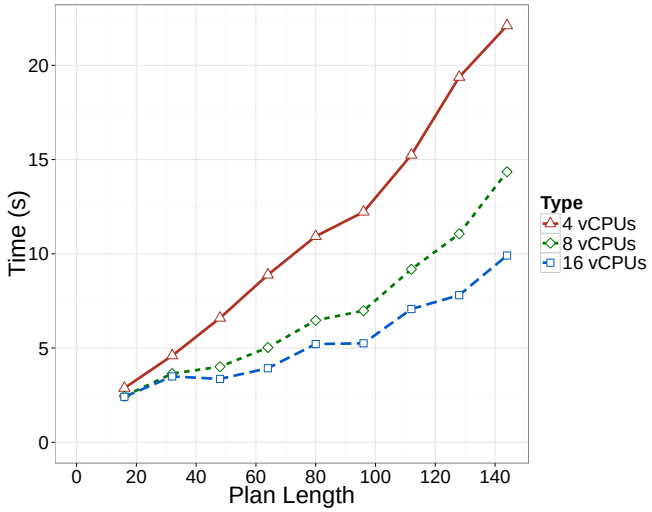


Fig. 6: Runtime of the SR strategy, on a *linear scale* using VMs with a varying number of vCPUs.

increases, ICR and SR offer more and more improvement over MCR in terms of plan generation time.

While the ICR strategy uses all available vCPUs to generate plans using intermediate checkpoints, SR allocates one vCPU for MCR and the remaining ones for ICR. Therefore the ICR approach finds plans more quickly. Due to the fact that a vCPU is only a hyperthreaded core, and due to the observations from Table 1, it is hard to hypothesize on the exact effect of the MCR computation on the parallel ICR threads. From the data, it appears that ICR takes typically 20-30% less time than SR, once an intermediate checkpoint has been created.

5.1.2 Scalability

To measure how well SR can scale when given more compute power, we ran the experiments where we varied plan length and the machine type. As stated, we used AWS C4 instances with 4, 8, and 16 vCPUs. The results of this experiment are shown in Figure 6.

We observe that plan generation time improves strongly when the number of vCPUs is increased from 4 to 8. Similarly, when the number of vCPUs is increased from 8 to 16, plan generation time speeds up by a similar factor ($\sim$1.5).

While additional CPU cores can parallelize the planning processes, the centralized overhead (reading files, generating planning problems, managing threads, and aggregating the results) attributes for an increasing runtime for rollback plan generation. Our undo system prototype is not fully optimized, so we believe more performance gains can be realized by careful performance tuning. Also, optimizing checkpoint creation time can improve usability of the system. In the above experiments, checkpoint creation time varied between 40 and 70 seconds; checkpoint creation time was affected by the responsiveness of EC2 APIs and the number of resources.

Three points should be noted. First, while our parallelizing strategies, SR and ICR, can benefit from additional cores, MCR cannot. Second, the approach of using intermediate checkpoints is not generally applicable, but specifically relies on the undo scenario where "do"-actions can be monitored and intermediate checkpointing can be realized.

Finally, neither SR nor ICR is geared to finding the shortest rollback plan. In the experiments here, this has no effect since all strategies lead to the same plans as per our experiment design. In general, though, the rollback plans generated using *Intermediate Checkpoint Rollback* are not guaranteed to be optimal in terms of plan length. For example, consider a case like Scenario 3) in Section 2, where an administrator executes a large number of commands and reaches a state which can be rolled back by a small number of steps. Rollback plan generation using the ICR approach will track all commands that change the state of the system, consider all intermediate checkpoints, and likely generate a sub-optimal plan. In contrast, the MCR strategy typically generates an optimal plan.

5.2 Anytime Rollback

As discussed above, ICR and SR improve the undo plan generation time, but do not take plan length into account. MCR does, but suffers from the exponential complexity of the planning problem. Anytime Rollback (AR) provides a flexible middle ground: through ICR, it computes a first plan as quickly as possible, and then iteratively improves plan quality. Since AR is a timeout based algorithm, it cannot be directly compared with SR. Therefore, we design experiments where we can measure and visualize improvements within specified timeout.

5.2.1 AR Experiment Setup

To demonstrate that AR improves plan quality, we have to consider scenarios where the following conditions are true:

$$quality(MCR) \geq quality(ICR) \quad (1)$$

$$comp.time(MCR) = comp.time(ICR) + \epsilon \quad with\ \epsilon > 0 \quad (2)$$

There are three scenarios that fit these criteria:

S1 - ICR is optimal. In this case, AR returns optimal solution after a timeout period, and does not improve plan length if the timeout is increased.

S2 - ICR is sub-optimal. In this case, ICR produces sub-optimal plans but AR can improve the results. Plan quality can be improved by increasing timeout period until the optimal solution is found. However, MCR is not the only optimal solution.

S3 - MCR is the only true optimal solution. In this case, the best plan can only be generated using MCR, and ICR is always sub-optimal. Plan quality can be improved using the Anytime Rollback algorithm. However, the optimal plan will be returned only if MCR runs to completion before the specified timeout is reached. We *focus the evaluation of AR on this scenario,* because we can visually show AR's solution improvement over time.

5.2.2 Experiment Design

To demonstrate quality improvement offered by AR, we design planning problems that represent checkpoints in a 2D plane of squares. However, we do not utilize the 2D plane in planning, since this would pose an unfair advantage over typical real-world planning solutions, such as undo for AWS EC2. The 2D plane allows us to position checkpoints (MC, IC, CC) in well-defined relation to each other and to visualize planes in a human-understandable way.

We design a domain where locations are represented by integer Cartesian co-ordinates, and actions are moves to go up, down, left, or right by one unit. The distance between two locations is thus the Manhattan distance, the sum of the absolute difference of their co-ordinates: the distance between two points (x_1, y_1) and (x_2, y_2) is $d = |x_2 - x_1| + |y_2 - y_1|$.

In this domain, we create checkpoints based on location represented by Cartesian co-ordinates. Given a Manual Checkpoint (MC) at (x_0, y_0), Intermediate Checkpoints (ICs) are created equidistant from each other, with distance n as before, until the Current Checkpoint (CC) at (x_c, y_c) is reached. A rollback plan is a sequence of actions that forms a path from CC to MC. The goal of AR is to produce the shortest path from (x_0, y_0) to (x_c, y_c) it can find within a specified time.

To port this domain into the planning world, we map it to the Blocksworld [14] domain, which is well-known in AI planning. Theoretically, we could model Cartesian Co-ordinates using PDDL numeric expressions. However, such a model would be fundamentally different from domains like AWS, would allow numeric heuristics, and require a specialized planner – all of which would severely limit the generalizability of the experiments. In contrast, Blocksworld domain is similar to AWS because we can represent Cartesian Coordinates using different types of objects and their state. Also, we can observe exponential nature of planning on relatively short plan lengths.

Objects in the Blocksworld domain are stacked blocks. A block is either *clear* or there is another block sitting on top of it. A clear block b can be moved from one stack s_1 to another stack s_2 with two actions: *unstack*(b, s_1) and *stack*(b, s_2).

In our mapping, each checkpoint has four stacks: $X_{current}$, $Y_{current}$, X_{temp}, and Y_{temp}. Blocks on *current* stacks represent Cartesian co-ordinates of the current position. *temp* stacks are simply placeholders for blocks. Our AI planning problem formulation uses *temp* stacks so that the planner can change current coordinates by moving blocks from *current* stacks and placing it on *temp* stacks.

The objective for the planner is to move blocks from *current* stacks to *temp* stacks or vice versa, so as to achieve moving the current position from the initial state to the goal state. Figure 7 shows a transition from position $(3, 2)$ to $(1, 1)$. Since blocks are moved from top of *current* stacks, the order of blocks is opposite in *temp* stack.

Many planning tasks do not need to move all blocks from *current* stack. In such scenarios, we can prune *current* stack by ignoring such blocks, and reduce the problem size.

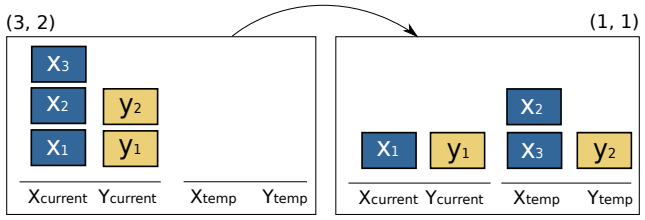


Fig. 7: A planning task from (3, 2) to (1, 1) in Blocksworld domain. The Cartesian coordinates of the current position are represented by *current* stacks. Blocks are moved from *current* to *temp* or vice versa to transition between states.

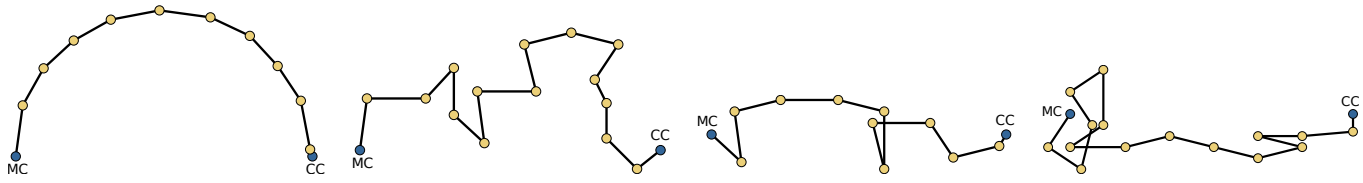
We designed four scenarios where ICR finds a plan faster than MCR, but MCR is the only true optimal solution. These scenarios are shown in Figure 8.

The Semicircle scenario is designed to be symmetrical. Despite having multiple edge exploration strategies, most edges, especially those near the circumference, do not improve the rollback plan significantly. In contrast, the other three asymmetric scenarios were created with some randomness. Therefore, for each of these scenarios we expect different quality improvements. All scenarios have an optimal plan length of 256 steps, and the Initial and Goal checkpoints are placed on a straight line so that there is no variance in complexity of planning task for MCR.

We encoded the scenarios into Blocksworld as per above. To measure solution quality over time, we set the timeout to infinity, i.e., we wait for MCR to find the optimal solution, and observe AR's improvements until then.

For the experiments, we used a cluster of Amazon EC2 C4.large instances, where each instance has 3.75 GB memory and 2 vCPUS. The cluster is located in a single Availability Zone in AWS, and communication between master and workers is minimal. We do not expect bandwidth to be a bottleneck but latency can have some impact. However, we did not study any effects of the network separately, these are included in the overall planning time. In these experiments, we measured *total* time taken for generating and aggregating plans.

In the following experiments, we show that despite communicating over a network, Anytime Rollback can improve plan quality within a short time. First, we demonstrate plan quality improvement in different scenarios. Then we analyse the results and choose a particular configuration for further analysis. Finally, we describe and evaluate various algorithms that facilitate these improvements.



(a) Semicircle Path (b) Custom Path 1 (c) Custom Path 2 (d) Custom Path 3

Fig. 8: Selected scenarios for Anytime Rollback (AR)

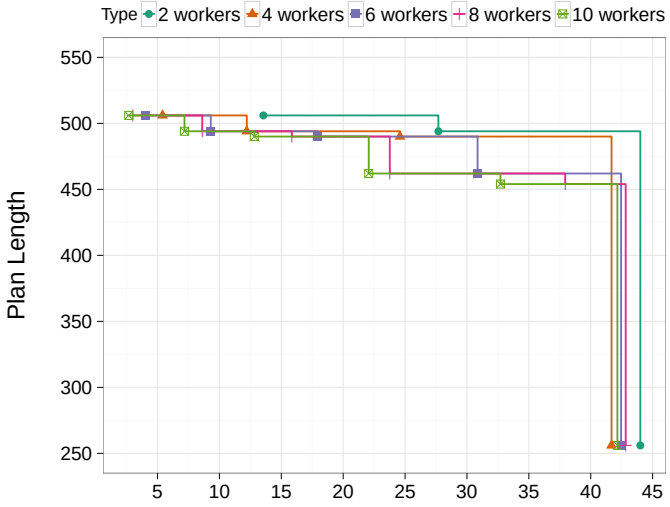


Fig. 9: Performance of AR with 10 worker nodes and checkpoint distance $n = 50$ using Depth First edge exploration. Quality improvement is shown as step chart. AR runs to completion, i.e., when Plan Length reaches the optimal value of 256 steps.

5.2.3 Plan Improvement

We first report on measurements of AR's performance with the Depth First edge exploration algorithm in the semicircle scenario. In this scenario, intermediate checkpoints (ICs) were created along a semicircular path from CC to MC at an IC distance of n.

We ran AR on clusters of different sizes and for different values of n. For each setting, we ran AR 15 times and calculated median times for each plan improvement. We then plotted a step chart, such as the one shown in Figure 9. Better performing configurations generate a first plan earlier and achieve a quicker decrease in plan length over time, and thus cover a smaller *area under the curve*. Therefore, we generated a step chart for each configuration and calculated the area under the curve as follows: we used the y-axis as the basis of the curve i.e. we calculated the area of curve that covered the region between $(0, y(\text{first plan}))$ and $(0, y(\text{optimal plan}))$ on the y-axis. Since $y(\text{first plan})$ is determined by ICR and $y(\text{optimal plan})$ by MCR, the values are the same for all AR algorithms. However, the time varies across the algorithms, and the curve is undefined for $x < x(\text{first plan})$. By limiting the calculation as above, we essentially take the value of $y(\text{first plan})$ for $0 \leq x < x(\text{first plan})$, and thus achieve better comparability. To further ease comparability, we calculated the areas *relative* to the minimum area in the set of configurations to be compared. The results for the Depth First strategy on the semicircle are shown in Table 2.

Workers	Checkpoint Distance n	Area under curve / min. area under curve
10	10	1.044
	40	1.046
	50	1
8	10	1.040
	40	1.049
	50	1.027
6	10	1.040
	40	1.073
	50	1.043
4	10	1.038
	40	1.093
	50	1.067
2	10	1.081
	40	1.167
	50	1.16

TABLE 2: Area analysis for varying checkpoint distance and workers (Depth First edge exploration, semicircle scenario)

As can be seen from the table, the configuration with 10 workers and $n = 50$ has the minimum area under curve. However, the improvements are arguably not huge in this setting and may not justify the additional cost. Figure 9 shows the performance profile of different of different cluster configurations where $n = 50$. We can observe that the configuration with 10 workers generates a plan faster and shows better plan improvements over time than the other scenarios. Therefore, we use this configuration for further experiments.

We conducted a further analysis between checkpoint distance, number of workers, and area under the curve. Our analysis showed no conclusive relationships in general. One aspect of practical importance, however, is that in scenarios with 2-4 workers a checkpoint distance of 10 steps led to better results. Thus, if compute budget is a major consideration, this checkpoint distance might be more suitable.

5.2.4 Comparing Edge Exploration Algorithms

To measure the performance of different edge exploration algorithms, we performed 15 runs of AR on 10 different scenarios (with 10 workers and checkpoint distance of 50). Figure 8 shows four of the 10 scenarios: the "semicircle" and three different random paths. We created 9 such random paths. For objective comparison of the 10 scenarios, we calculated the area under curve for each algorithm in each of the 10 scenarios, where smaller is better. For each scenario, we calculated the area relative to the minimum found in that scenario. The results of this analysis are shown in Figure 10.

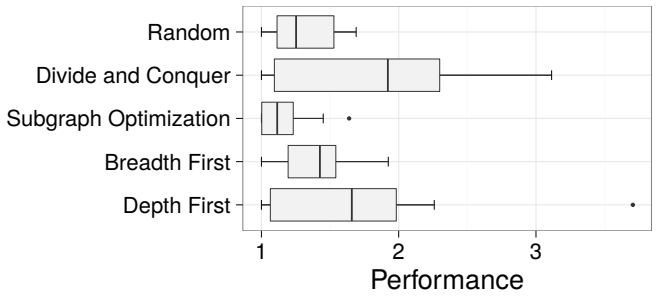


Fig. 10: Relative performance of edge addition algorithms in 10 rollback scenarios. Area under curve relative to minimum of each scenario is shown in standard IQR based box plot.

The best value that an algorithm can achieve for a particular scenario is 1. We can observe that Subgraph Optimization algorithm performs better overall. Also, on average, performance of Random algorithm is better than Breadth First, Depth First, and Divide and Conquer algorithms.

For a more in-depth view, the results for the four scenarios from Figure 8 are shown in Figure 11. As before, the curves depict the median over 15 runs, except for Random edge exploration: since it is non-deterministic, we plot the best and worst cases out of the 15 runs.

We can observe that, in the semicircle scenario, plans improve slowly but steadily. This may stem from the symmetric design. In contrast, for non-symmetric scenarios, improvement in first few seconds is often dramatic.

Rollback plans of AWS domain cannot be translated to strict 2D representation. Therefore, we do not draw conclusions about the strengths and weaknesses of these algorithms beyond the empirical evaluation presented.

6 Discussion

The results shown here are highly encouraging with respect to the direction of parallelizing the planning task using intermediate checkpoints. Comparing the most extreme case of SR, i.e., rollback plan of length 144, MCR takes 394.08 seconds as opposed to 9.91 seconds for SR on a 16vCPU machine. While the comparison of MCR on a 4vCPU machine with SR on a 16vCPU machine has to be taken with a grain of salt – see the discussion below – this corresponds to a *speedup of factor 39.76.*

In Section 4.4.1, we show that all planning processes tend slow down when run in parallel. Despite this behavior, ICR and SR significantly outperform MCR. Similarly, Distributed Anytime Rollback (AR) improved plan quality over time in our tested scenarios. In particular, AR showed significant plan improvement in the first 1/5th of total planning time of MCR in non-symmetric scenarios.

We define plan quality in terms of plan length. Since it is possible for longer plans to have shorter execution times, this speedup may not improve execution time proportionally. Also, we do not make any assumptions about the hosted applications, their state, and how infrastructure level configuration changes affect them. Using planning algorithms that consider application-level constraints or dimensions like estimated execution times or cost of operations would be interesting directions for future work. For instance, execution times or costs could be modeled as weights in the planning graph, and using a single such dimension might be straight-forward. Considering multiple dimensions may require more elaborate extensions.

By circumventing the need for solving the whole end-to-end planning task, we avoid the exponential behavior of AI planning and achieve roughly linear scalability. With the addition of the Distributed Anytime Rollback algorithm, we enable a flexible trade-off between plan quality and computation time. There are, however, limitations and threats to validity in our study, which we will discuss next.

6.1 Internal Validity

In our experiments, the memory size is larger for machines with a larger number of CPU cores. Plan generation time may or may not be affected by this factor – the planner might not use the additional memory. We are, as yet, not certain regarding the extent of the effect.

Also, our experiments were run on virtual machines beyond our direct control. Computation on resources that are potentially shared with other users of the cloud may be subject to significant performance variation, and even machines of the same type are known to have fluctuating performance [16]. We aimed to minimize this factor through the use of C4 machines, which offer more reliability in terms of compute performance. However, such variations cannot be ruled out in virtual infrastructure with multi-tenancy.

6.2 External Validity

In our experiment scenarios, we performed a subset of available actions (attach, detach, start, and stop). We cannot guarantee that plan generation times will be similar when a different set of actions is performed. Furthermore, for AR we distributed the computation of rollback plans across a network. A user's network may introduce latency or outages that we did not account for in our studies.

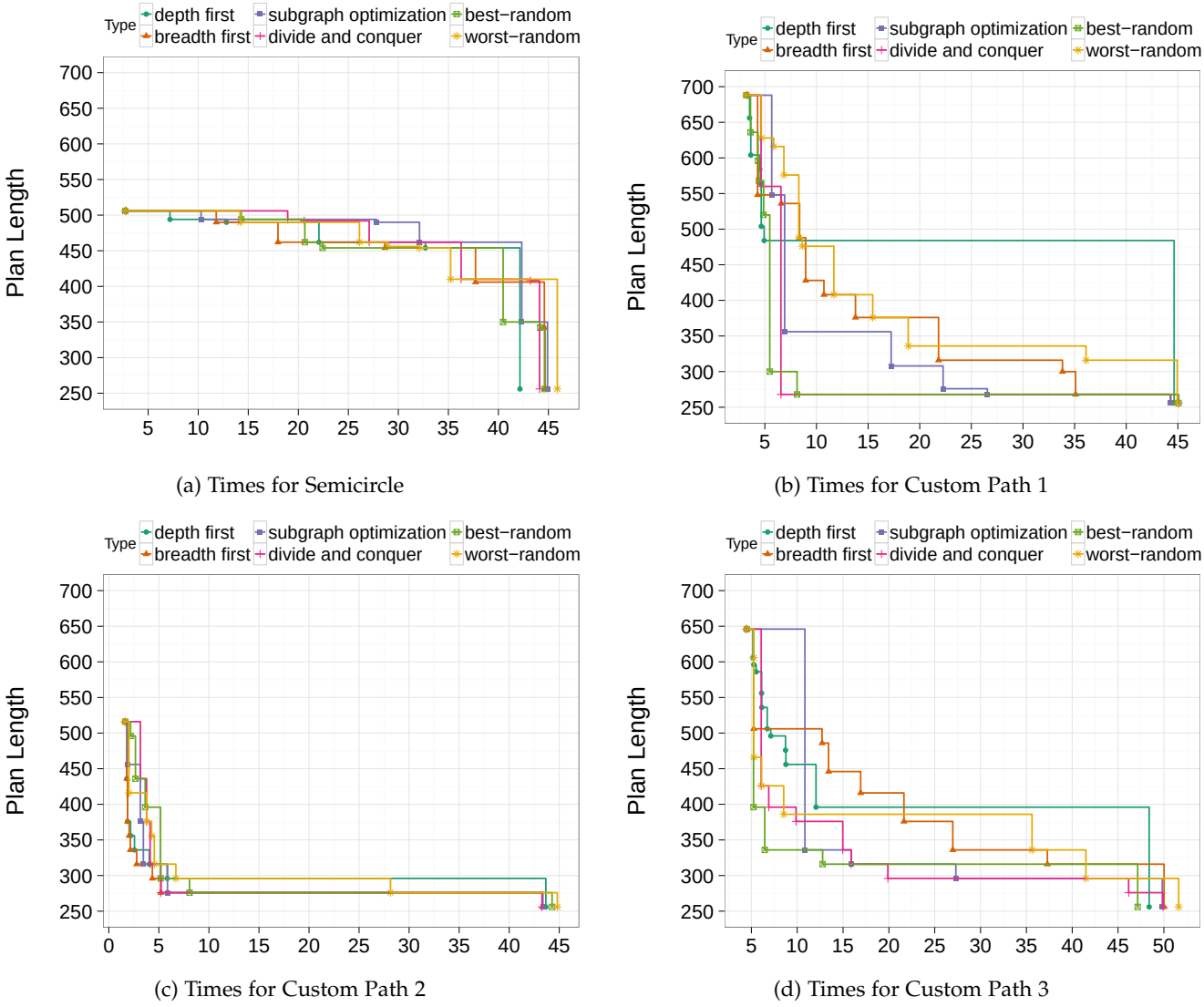
We assume that planning tasks are not executed on the application servers or virtual machines that are subject to configuration changes. Therefore, in our experiments, planning is not affected by application workload. We cannot guarantee that similar plan generation times can be achieved if application servers are used.

6.3 Transferability of AR results to AWS

We studied AR with a Blocksworld encoding and problems that have an optimal solution of 256 steps. This may not reflect AWS domain, and therefore Anytime Rollback may show different plan improvement characteristics for AWS.

7 Related Work

Rollback methods can be checkpoint-based [17], [18], log-based [17], [19], or shadow page-based [19]. Our approach is checkpoint-based, where checkpoints store relevant information on the disk. In our approach, rollback means achieving physical state of cloud resources that match the state stored in checkpoint. In contrast to other checkpoint based approaches, where rollback is preformed by copying saved information back to memory, our rollback operation



(a) Times for Semicircle

(b) Times for Custom Path 1

(c) Times for Custom Path 2

(d) Times for Custom Path 3

Fig. 11: Plan Length improvement with AR on various scenarios.

requires execution of API operations. This is similar to *Sagas* [20], where system designers provide compensating actions for each operation, so that undo operations would require execution of corresponding compensation actions in reverse order. However, as argued in Sections 1 and 2, undoing actions in reverse chronological order can be highly suboptimal.

AI planning has been used for system configuration, e.g. [21]–[27], and cloud configuration, e.g. [28]–[30]. In [29] planning is applied in a straight-forward fashion to the problem of reaching user-specified goal. The work is integrated into cloud management tools such as Facter, Puppet, and ControlTier. In [30] hierarchial task network (HTN) planning is applied at PaaS level for fault recovery. Similarly, [28] uses HTN planning to achieve configuration changes for system modeled in common interface model (CIM) standard. Besides AI planning, POMPDPs (Partially Oservable Markov Decision Processes) can be used for recovery process [31]. Using intermediate checkpoints to speed up planning for undo has not been the subject of any works we are aware of.

In [9], so-called Status and Action Management (SAM) models which describes Business Object behaviors are automatically translated to PDDL domain models. Since a formal model comparable to SAM was not available to us for AWS EC2 [3], we have manually modeled key parts of the AWS API in PDDL. In [4], AI planning is used to generate rollback plans on cloud operations. Our work extends this approach by dividing and parallelizing planning processes.

8 Conclusion

To reduce the impact of human-induced errors in API-controlled cloud environments, it can be highly beneficial to provide rollback of inadvertent changes made by an administrator. Our prior approach works well for rollback with a small number of actions, but does not scale as the number of actions increases.

In this paper, we present a scalable approach to rolling back cloud operations using AI planning. Our undo system intercepts API calls, creates checkpoints, uses an AI planner

to generate a plan that takes the system back to previous state, and finally executes the plan.

We discuss four strategies to generate rollback plans. Through experiments we demonstrate that, compared to the naïve approach, our Scalable Rollback approach improves plan generation time as plan length increases. For the most extreme case we tested, we observe speedup by a factor of nearly 40. In our experiments with the Blocksworld domain, we also demonstrated that the Distributed Anytime Rollback algorithm offers a trade-off between plan quality and computation time.

These improved algorithms should enable the use of undo systems in most practical cases, since the usual practical limits stemming from AI planning complexity can be circumvented for undo.

References

[1] D. Oppenheimer, A. Ganapathi, and D. A. Patterson, "Why do internet services fail, and what can be done about it?" in *USITS'03: USENIX Symposium on Internet Technologies and Systems*, 2003.
[2] J. Gray, "Why do computers stop and what can be done about it?" in *Symposium on Reliability in Distributed Software and Database Systems*, 1986, pp. 3–12.
[3] "AWS Elastic Compute Cloud (EC2)," http://aws.amazon.com/ec2/, accessed: 2015-06-02.
[4] I. Weber, H. Wada, A. Fekete, A. Liu, and L. Bass, "Automatic undo for cloud management via AI planning," in *Proc. USENIX Workshop on Hot Topics in System Dependability*, 2012.
[5] T. Bylander, "The computational complexity of propositional strips planning," *Artificial Intelligence*, vol. 69, pp. 165–204, 1994.
[6] S. Satyal, I. Weber, L. Bass, and M. Fu, "Scalable rollback for cloud operations using AI planning," in *Software Engineering Conference (ASWEC), 2015 24th Australasian*. IEEE, 2015, pp. 195–202.
[7] I. Weber, H. Wada, A. Fekete, A. Liu, and L. Bass, "Supporting undoability in systems operations," in *USENIX LISA'13: Large Installation System Administration Conference*, Nov. 2013.
[8] "AWS CloudTrail," http://aws.amazon.com/cloudtrail/, accessed: 2015-06-02.
[9] J. Hoffmann, I. Weber, and F. M. Kraft, "SAP speaks PDDL: Exploiting a software-engineering model for planning in business process management," *Journal of Artificial Intelligence Research (JAIR)*, vol. 44, pp. 587–632, 2012.
[10] J. Hoffmann and B. Nebel, "The FF planning system: Fast plan generation through heuristic search," *J. Artif. Int. Res.*, vol. 14, no. 1, pp. 253–302, May 2001.
[11] D. McDermott *et al.*, *The PDDL Planning Domain Definition Language*, The AIPS-98 Planning Competition Comitee, 1998.
[12] T. L. Dean and M. S. Boddy, "An analysis of time-dependent planning." in *AAAI*, vol. 88, 1988, pp. 49–54.
[13] S. Zilberstein and S. Russell, "Approximate reasoning using anytime algorithms," *KLUWER INTERNATIONAL SERIES IN ENGINEERING AND COMPUTER SCIENCE*, pp. 43–43, 1995.
[14] N. Gupta and D. S. Nau, "On the complexity of blocks-world planning," *Artificial Intelligence*, vol. 56, no. 2, pp. 223–254, 1992.
[15] M. Helmert, "The fast downward planning system," *CoRR*, vol. abs/1109.6051, 2011. [Online]. Available: http://arxiv.org/abs/1109.6051
[16] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz, "Runtime measurements in the cloud: observing, analyzing, and reducing variance," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, 2010.
[17] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Comput. Surv.*, vol. 34, no. 3, pp. 375–408, Sep. 2002.
[18] B. Randell, "System structure for software fault tolerance," *IEEE Transactions On Software Engineering*, vol. 1, no. 2, 1975.
[19] G. Graefe, "A survey of B-tree logging and recovery techniques," *ACM Trans. Database Syst.*, vol. 37, no. 1, pp. 1:1–1:35, Mar. 2012.
[20] H. Garcia-Molina and K. Salem, "Sagas," in *SIGMOD'87: Proc. Intl. Conf. on Management Of Data*. ACM, 1987, pp. 249–259.
[21] N. Arshad, D. Heimbigner, and A. L. Wolf, "Deployment and dynamic reconfiguration planning for distributed software systems," in *IEEE Conf. on Tools with Artificial Intelligence*, 2003, pp. 39–46.
[22] N. Arshad, D. Heimbigner, and A. Wolf, "A planning based approach to failure recovery in distributed systems," in *WOSS'04: ACM SIGSOFT Workshop on Self-Managed Systems*, 2004, pp. 8–12.
[23] A. J. Coles, A. I. Coles, and S. Gilmore, "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.
[24] C. E. da Silva and R. de Lemos, "A framework for automatic generation of processes for self-adaptive software systems," *Informatica*, vol. 35, pp. 3–13, 2011, publisher: Slovenian Society Informatika.
[25] F. Dalpiaz, P. Giorgini, and J. Mylopoulos, "Adaptive socio-technical systems: a requirements-driven approach." *Requirements Engineering*, 2012, springer, to appear.
[26] B. Drabble, J. Dalton, and A. Tate, "Repairing plans on-the-fly," in *Proc. of the NASA Workshop on Planning and Scheduling for Space*, 1997.
[27] K. Levanti and A. Ranganathan, "Planning-based configuration and management of distributed systems," in *Proceedings of the 11th IFIP/IEEE international conference on Symposium on Integrated Network Management*, ser. IM'09. IEEE Press, 2009, pp. 65–72.
[28] S. Hagen and A. Kemper, "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*, ser. CLOUD '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 11–18.
[29] H. Herry, P. Anderson, and G. Wickler, "Automated planning for configuration changes," in *LISA'11: Large Installation System Administration Conference*, 2011.
[30] F. Liu, V. Danciu, and P. Kerestey, "A framework for automated fault recovery planning in large-scale virtualized infrastructures," in *MACE 2010, LNCS 6473*, 2010, pp. 113–123.
[31] K. R. Joshi, W. H. Sanders, M. A. Hiltunen, and R. D. Schlichting, "Automatic model-driven recovery in distributed systems," in *SRDS'05: IEEE Symposium on Reliable Distributed Systems*, 2005.

Biographies

Suhrid Satyal is a PhD Candidate at University of New South Wales (UNSW), Sydney. He is also affiliated to Architecture & Analytics Platforms (AAP) team at Data61, CSIRO in Sydney, Australia. He received his BSE in Information and Communication Technologies from the Asian Institute of Technology (AIT), Thailand. His main research interests are Business Process Management, DevOps, Web Applications, and Cloud Computing.

Ingo Weber is a Principal Research Scientist & Team Leader of the Architecture & Analytics Platforms (AAP) team at Data61, CSIRO in Sydney, Australia. In addition he is an Adjunct Associate Professor at Swinburne University, Melbourne, and an Adjunct Senior Lecturer at the University of New South Wales (UNSW), Sydney. He has published over 80 refereed papers and two books. Ingo has served as PC member for WWW, BPM, ICSOC, AAAI, and IJCAI, etc. Prior to Data61/NICTA, he worked at UNSW and at SAP Research Karlsruhe, Germany.

Len Bass is the coauthor of two award-winning books in software architecture: *Software Architecture in Practice* and *Documenting Software Architectures: Views and Beyond*, as well as several other books and numerous papers in computer science and software engineering including his latest book, *DevOps: A Software Architect's Perspective*. Len has over 50 years experience in software development, 25 of those at the Software Engineering Institute. He also worked for three years at National ICT Australia Ltd. (NICTA) and is currently an adjunct faculty at Carnegie Mellon.

Min Fu is a research fellow in the department of computing, Macquarie University, Australia. He is also a visiting scientist in Data61, CSIRO in Sydney, Australia. He received his PhD degree from the University of New South Wales (UNSW), Sydney. His research interests include: cloud computing, data analytics, machine learning, data science, cyber security, software architecture, dependable computing, software engineering. He has over 4 years' research experience in computer science, and 5 years' work experience from IT industry.