Detecting Cloud Provisioning Errors Using an Annotated Process Model

Xiwei Xu[1], Ingo Weber[1,2], Len Bass[1,2], Liming Zhu[1,2], Hiroshi Wada[1,2], Fei Teng[1]
[1]NICTA, Sydney, Australia
[2]School of Computer Science and Engineering, University of New South Wales, Sydney, Australia
{firstname.lastname}@nicta.com.au

ABSTRACT
In this paper, we demonstrate the feasibility of annotating a process model with assertions to detect errors in cloud provisioning in near real time. Our proposed workflow is: a) construct a process model of the desired provisioning activities using log data, b) use the process model to determine appropriate annotation triggers and annotate the process model with assertions, c) use the process model to monitor the deployment logs as they are generated, d) trigger the assertion checking based on process activities and log entries, and e) check the assertions to determine errors.

For a production deployment tool, Asgard, we have implemented the steps involving constructing a process model, using the model to determine appropriate annotation triggers, triggering the annotation checking based on Asgard log files, and detecting errors. Our prototype has detected errors that cross deployment tool boundaries and go undetected by Asgard; it further has detected other errors substantially more quickly than Asgard would have.

Categories and Subject Descriptors
D.2.4 [Software Program Verification] reliability

General Terms
Reliability

Keywords
System administration, deployment, cloud provisioning, error detection

1. INTRODUCTION
Deploying applications in cloud environments introduces uncertainties for operations that have traditionally been under the direct control of an enterprise. Enterprises become dependent on the cloud infrastructure to provision resources. The uncertainty arises from the inherent randomness in the behavior of cloud environments, caused by day-to-day node and instance failures, rare large-scale disasters, workload spikes, and the like. In addition to the uncertain cloud environment, configuration errors cause a significant fraction of system failure. Some configuration errors are subtle and take a long time to detect and diagnose, thus leading to long recovery time [1]. Indeed, Matt Welsh has called the problem "configuration hell"[1].

MW4NG'13 December 9–13, 2013, Beijing, China.
ISBN: 978-1-4503-2551-6

Detecting errors occurring during the provisioning (configuring and deploying) of cloud applications is a difficult process. Some of the reasons why provisioning is error prone are that multiple independent systems are involved and configuration specifications must be consistent, log files are voluminous and inconsistent in style, and particular sequencing of events must be enforced.

We propose using a process model annotated with assertions to detect provisioning and configuration errors in near real time. The process model provides specification of the order and parallelism possibilities of the events of the provisioning, the annotations provide specific assertions that can be checked when the provisioning process has reached particular steps, and the assertions are checked by examining the actual state of the deployment, not by inferring the state.

Figure 1 shows our proposed workflow for accomplishing this error detection. Figures 1a and 1b represent what happens offline prior to the actual provisioning and Figure 1c represents the online evaluation.

Figure 1a shows the creation of the process model. This is accomplished by using logs created by successful provisioning. This process is semi-automatic in that the steps of the process model must be given meaningful names and must be at the correct granularity.

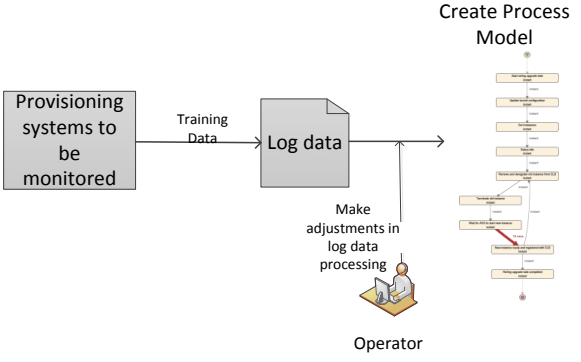

Figure 1a. Creating the process model.
(Refer to figure 4 for the detailed process model.)

Figure 1b shows the annotations being added to the process model and the associated triggers and assertions being developed. This activity is guided by the steps of the process model and their granularity. Locations for annotations are, typically, the beginning or the end of a process step. It is also possible to have annotations during a process step; however, annotations that are placed during a process step cause consideration of the granularity of the particular process step.

[1] http://matt-welsh.blogspot.com.au/2013/05/what-i-wish-systems-researchers-would.html

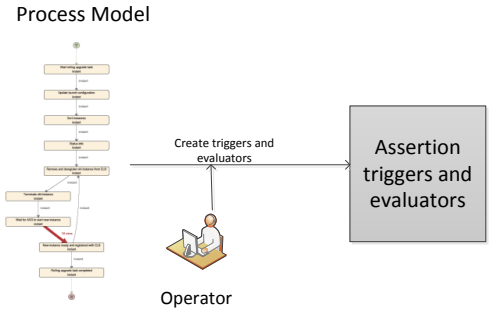

Figure 1b. Creating the assertions and their evaluators. (Refer to Figure 4 for the detailed process model.)

Figure 1c shows the monitoring of the log files while the provisioning is underway. The logs produced by the provisioning systems are placed in a central repository and this repository is used to trigger the various assertion evaluators.

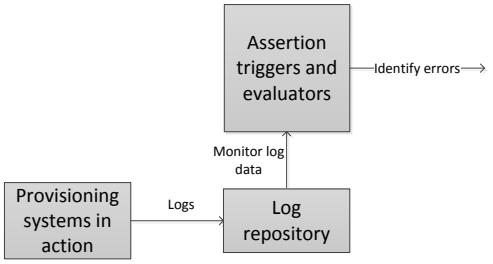

Figure 1c. Online use of assertion triggers and evaluators.

Note that the offline portion of this workflow needs to be executed just once for each combination of provisioning tools. Subsequently each use of those provisioning tools will execute the same process and, consequently, can be tested by the same set of triggers and evaluators.

Theoretically speaking, we take a holistic view of the provisioning process as involving multiple tools, events, and states that can be captured in a process model and tested by assertions. This view is new and is the contribution of this paper. Our approach can detect errors at run time, which is difficult due to the uncertain cloud infrastructure. Even with a "perfect" static specification, runtime failures can occur and these need to be detected (and recovered, if possible). In addition, our approach can detect errors that cut across systems, rather than errors specific to a single system.

To generalize our approach, we are investigating its application in other contexts. In terms of cloud environments, we are looking at VMware[2]. In terms of automated operation, we are looking at Chef[3]/Puppet[4]. In terms of our methodology, we are working on decoupling the creation of the process models from logs.

Practically speaking, we also report on the implementation of some of the steps of this workflow for the product deployment tool Asgard[5]. We report on the creation of the process model and the structure and use of the assertion triggers and evaluators. We created the triggers based on the errors we encountered while creating the training data for the process model creation. Our prototype implementation discovers some errors that Asgard does not detect and some errors much more quickly than Asgard would detect them. We begin by discussing related work, followed by the creation of the process model, the structure of the triggers and evaluators, and the use of our prototype.

2. RELATED WORK

Provisioning processes are often implemented in specific deployment tools (e.g., the Asgard tool used in this paper) or customized scripts (shell-based or using Chef/Puppet-like generic tools). Test-driven sys-admin frameworks, such as Chef-Cucumber[6], allow intermediary assertions to be written and integrated with the provisioning scripts. These approaches require instrumentation, synchronously waiting for the assertion evaluation – thus slowing down the provisioning process while not being integrated with external monitoring tools. Other approaches, such as WiDS [2], interpose and check the expected internal states during runtime. They focus on the state of the system rather than the processes and events leading to the state. The approach in [3] uses discrete event simulation to simulate the operation process and analyze the tradeoffs between different strategies resources to error detection and repair. FATE [4] allows the specification of events, can derive facts from events and compare facts with expectations. These approaches are used for internal protocols in distributed software and require in-depth understanding of the internals as well as instrumentation. Our approach uses logs to understand the processes and asynchronously evaluate the intermediary assertions in non-intrusive ways. This is particularly true for deployment tools that cannot be instrumented and where the logs are the only accessible artifacts during deployment.

Our process discovery technique from log files builds on the techniques from process mining – see the book by van der Aalst [5] for an overview. Process mining has been used to find errors in business processes [5] as well as to predict errors in hardware [6]. None of the uses of process mining has been to provide a description that can be annotated with assertions.

3. PROCESS MODEL CREATION

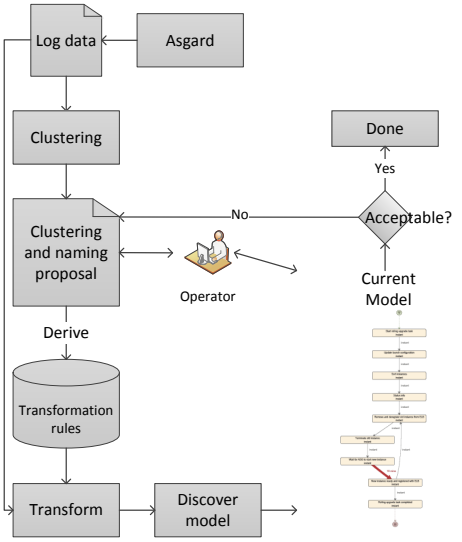

Figure 2. Creating the process model (Refer to Figure 4 for the detailed process model.)

The process model was created following the procedure in Figure 2. The log data reflects two successful executions of Asgard that

[2] http://www.vmware.com
[3] http://www.opscode.com/chef
[4] http://puppetlab.com
[5] https://github.com/Netflix/asgard

[6] http://www.cucumber-chef.org/

performed a rolling upgrade of a configuration with 4 instances. The first execution upgraded one instance at a time and produced a log file with 121 lines. The second execution upgraded two instances at a time and produced a log file with 253 lines.

The log files for the runs are then compared pairwise to create an NxN matrix. In our case, this was 374x374. Each cell of the matrix contains a value based on the Levenshtein distance [7] of the two lines in the log file used to index that cell. The Levenshtein distance measures the editing distance between two lines. We normalized this measure based on the length of the longer line.

For example, if we consider three log lines:

1. [2013-07-18 15:37:31,369] [Task:Pushing ami-ad059597 into group sdmcmm_appone--firstASG for app sdmcmm_appone] com.netflix.asgard.Task 2013-07-18_15:37:31 135: {Ticket: null} {User: null} {Client: localhost 0:0:0:0:0:0:0:1%0} {Region: ap-southeast-2} [Pushing ami-ad059597 into group sdmcmm_appone--firstASG for app sdmcmm_appone] Started on thread Task:Pushing ami-ad059597 into group sdmcmm_appone--firstASG for app sdmcmm_appone.
2. [2013-07-18 15:37:31,996] [Task:Pushing ami-ad059597 into group sdmcmm_appone--firstASG for app sdmcmm_appone] com.netflix.asgard.Task 2013-07-18_15:37:31 135: {Ticket: null} {User: null} {Client: localhost 0:0:0:0:0:0:0:1%0} {Region: ap-southeast-2} [Pushing ami-ad059597 into group sdmcmm_appone--firstASG for app sdmcmm_appone] Updating launch from sdmcmm_appone --firstASG-20130718120001 with ami-ad059597 into sdmcmm_appone --firstASG-20130718153731
3. [2013-07-18 15:37:31,998] [Task:Pushing ami-ad059597 into group sdmcmm_appone --firstASG for app sdmcmm_appone] com.netflix.asgard.Task 2013-07-18_15:37:31 135: {Ticket: null} {User: null} {Client: localhost 0:0:0:0:0:0:0:1%0} {Region: ap-southeast-2} [Pushing ami-ad059597 into group sdmcmm_appone--firstASG for app sdmcmm_appone] Create Launch Configuration sdmcmm_appone--firstASG-20130718153731' with image 'ami-ad059597'

We get the following symmetric matrix.

Table 1. Symmetric distance matrix.

0.	.711	.775
	0.	.593
		0.

The resultant NxN matrix is then clustered by a dendrogram creation tool, MultiDendrograms[7], using the "unweighted average" algorithm. This algorithm belongs to the family of Hierarchical Agglomerative Clustering (HAC) algorithms. A dendrogram yields a hierarchical arrangement of the source data where each node has a distance from its parent and children. Visualizing the hierarchy based on the distances provides a representation that shows groups of data items. The operator chooses a threshold for the clusters (a value that divides the data into coherent groups) and assigns each cluster a meaningful name.

Figure 3 shows the dendrogram derived from the log lines of our two executions of Asgard. The thick blue horizontal line is the line drawn by the operator to define meaningful clusters.

What we have done, thus far, is to use a similarity measure – the Levenshtein distance – and cluster similar log entries based on this measure. The last step in the clustering activity is to generate regular expressions. Each entry in the dendrogram represents a string taken from the log file. Each cluster is therefore a group of strings. A regular expression can be constructed such that it, more or less, represents the group of strings in a cluster. We used minimal acyclic DFAs [8] to construct these regular expressions.

The regular expressions we created are quite specific for these log lines. In the future, we expect to generalize the procedure for creating the regular expressions. The regular expression following this paragraph matches log line 2 above. Note that the tool replaces concrete numbers with "\d+" and concrete AMIs etc. with "ami-[0-9a-f]{8}", the latter portion meaning eight hex values. This is a domain specific encoding for AWS. The first 333 characters are replaced with ".{333}"since the first 333 characters are the same for all line entries in the log of one run, except for timestamps.

.{333} <whitespace> Updating <whitespace> launch <whitespace> from <whitespace> sdmcmm_appone--firstASG-\d+ <whitespace> with <whitespace> ami-[0-9a-f]{8} <whitespace> into <whitespace> sdmcmm_appone --firstASG-\d+

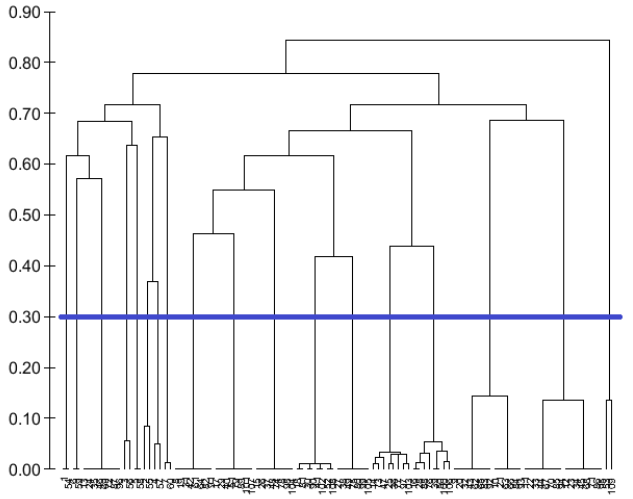

Figure 3. Dendrogram produced by MultiDendograms.

The named activity and the related set of log items are used as input into a process discovery tool, Disco[8] [5, Chap 5]. The output of the tool is the process model used to define the triggering activities during additional executions of Asgard.

The process model that was generated from our training set is shown in Figure 4. The names of the nodes are the names assigned by the operator and the labels on the lines are the time taken to make the transformation.

It is worth noting that prior to using a process-mining tool, we developed a process model manually by investigating the source code of Asgard. Our manual model was incorrect in several particulars, which we discovered from Figure 4 and verified by examining the particular log entries manually. The ground truth for a process model in our approach comes from execution, not from inspection.

4. CREATION OF ASSERTIONS AND EVALUATORS

According to the process model mined from Asgard logs, we manually created a set of assertions for each of the critical steps. We determined the critical steps based on our experience. Not every step is critical from a cloud provisioning perspective. For example, the third step in the process model of sorting instances, which sorts the instances in the ASG by their launch times, was not seen as

[7] http://deim.urv.cat/~sgomez/multidendrograms.php

[8] http://www.fluxicon.com/disco/

critical. The mapping between the operation steps and the corresponding assertions is shown in table 2.

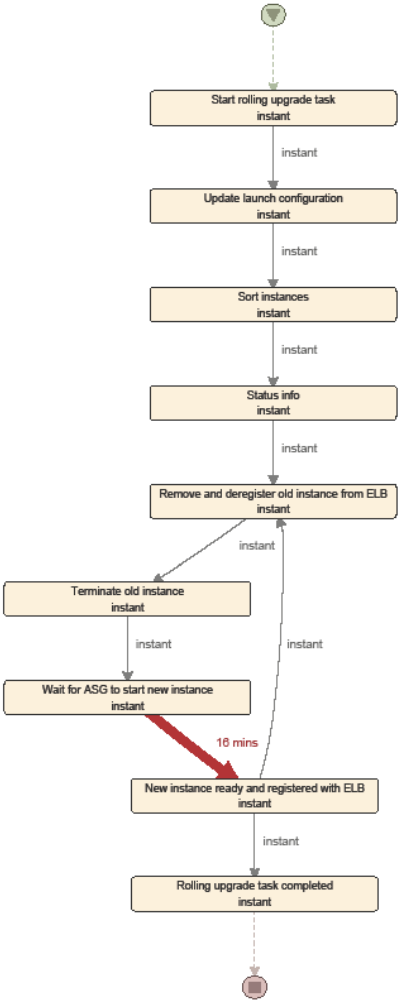

Figure 4. Output from Disco giving the process model.

Table 2. Assertions of Asgard process.

Steps	Assertions
Update launch configuration	1. ASG uses new launch configuration
Remove and deregister old instance (i) from ELB	2. i has been deregistered from ELB 3.i has been removed from ASG
Terminate old instance (i)	4. i is successfully terminated
Wait for ASG to launch new instance (i')	5. i' is successfully launched
New instance (i') ready and registered with ELB	6. i' has been registered to ELB 7. i' has been added to ASG

Each assertion is implemented by an evaluator, which could utilize Amazon Web Services (AWS) API (mainly describing different resources) directly or other third-party monitor frameworks to gather information to evaluate whether the assertion is true, as shown in Figure 5. We used Edda[9], an open-source solution to track infrastructure resources within AWS. Edda has better query ability than the AWS API and also supports querying the history of infrastructure resources. The evaluators are implemented as a set of RESTful Web Services based on Restlet[10] – a java RESTful framework.

5. STRUCTURE OF TRIGGERS AND EVALUATORS

Figure 1c shows the core idea of our assertion approach. We collect the logs from Asgard into a repository, which we monitor. Certain log lines trigger respective evaluators.

Specifically, we collect the Asgard logs using Logstash[11]. Logstash is an open source tool to collect, parse, index, and store logs. A Logstash agent periodically filters the noise elements of the log and looks for triggering conditions. Recall that the triggering conditions correspond to some location within the process model. When a triggering condition is found, the associated evaluator is invoked. The evaluator places information back into the Logstash repository where another Logstash agent examines it and sends information to a visualization component.

Figure 5 shows the various elements associated with this structure. There are three nodes in our environment, an operation node, an assertion evaluation node and a visualization node.

Asgard is deployed on the operation node within the cloud to manipulate infrastructure resources through the AWS API. The assertion evaluation node hosts a set of RESTful evaluation Web services to check assertions.

We deployed Logstash agents on the two nodes to collect the logs produced by Asgard and assertion evaluation services respectively and ship the logs to the database deployed on the visualization node. The Logstash agent on the operation node also pre-processes Asgard logs before shipping, including 1) filtering out noise, 2) aligning the log with the created process model through annotating log lines with process context as tags, 3) grouping log event with multiple log lines into a single entry, such as exception and errors, for better visualization, and 4) triggering assertion evaluation after each critical step.

The code snippet below gives two simplified sample records from Logstash. The first one is drawn from the original Asgard log. The original log content is shown in @message element, which says an instance is deregistered from an ELB (Elastic Load Balancer). Logstash uses regular expressions[12] to match the process relevant log lines, and annotate the matched log line with the label "asgard", as shown in @type element. The corresponding evaluator produces the second record, which has the label of "postcon" under @type element. The log states that the instance is successfully deregistered from the ELB. @tag element is used by Logstash for statistics and visualization. We added the same values to the @tag element of both records, so that the visualization node could associate the original log line with the corresponding evaluator log lines.

{"@source":"file://postcondition.com/asgard.log","@tags":["push","sdmcm
mapp--firstASG","step2"],"@timestamp":"2013-07-28T09:15:33.427Z",
"@source_host":"operation.com","@source_path":"//asgard.log","@messag
e":"[2013-07-28 19:15:33,187] [Task:Pushing ami-17cf5d2d into group

[9] https://github.com/Netflix/edda

[10] http://restlet.org

[11] http://logstash.net/

[12] Although, in principle, we could have used the process mining tool to detect which step individual log entries belong to, in our case, we detected step entry and exit by other means. In our future work, we plan to integrate the process mining and the triggering more closely.

sdmcmmapp--firstASG for app sdmcmmapp] Deregistered instances [i-e07969dd] from load balancer FirstELB ","@type":"asgard"}

{"@source":"file://postcondition.com/postcondition.log","@tags":["push","sdmcmmapp--firstASG","step2"],"@timestamp":"2013-07-28 T09:15:51.284Z","@source_host":"postcondition.com","@source_path":"//postcondition.log","@message":"[2013-07-28 19:15:50, 482] [postcondition] [Task:Pushing ami-17cf5d2d into group sdmcmmapp--firstASG for app sdmcmmapp] [Step:pushstep2] Instance i-e07969dd has been successfully deregistered from ELB FirstELB","@type":"postcon"}

6. PRELIMINARY EVALUATION

Our approach provides evaluator services in parallel with the deployment operation to (re-)evaluate the assertion after every critical operation step. The result of the evaluation is combined with the original operation logs to produce more evaluation information. In this section, we evaluate the proposed mechanism and additional evaluation information produced from an error detection perspective. We manually injected some cloud resource errors at different severity levels during rolling upgrades of an ASG with 4 instances, and use our prototype to detect those injected errors.

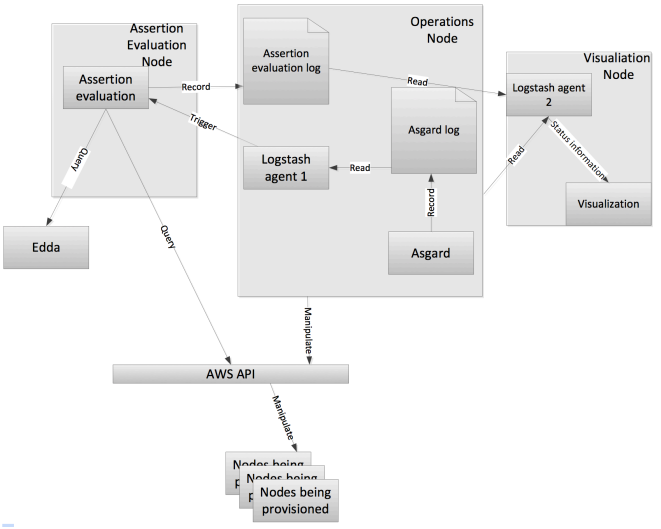

Figure 5. Structure of triggers and evaluators.

6.1 Detection of Temporary Erroneous State

The assertions we used for the basic checking are shown in table 2. Only the first assertion is evaluated once. All the other 6 assertions are evaluated for each pair of one old node and one new node. Thus, 25 assertions were evaluated in total, printing 25 extra log lines. The evaluation doesn't delay the deployment process of Asgard, because the assertions are evaluated asynchronously. Every evaluation is nearly instantaneous, except for short delays introduced by Logstash and Edda.

The system being deployed could be in a "temporary" erroneous state before returning to a correct state. Some of the temporary errors are detected and logged by Asgard, and some are not. We could detect such errors but it might not be valuable. In our first case, we injected two temporary errors: 1) deleting a new instance before it is ready, and 2) deleting a new instance after it is ready. Our evaluator detected both errors. Asgard, however, only detects the first error and launched another instance. For the second error, a compensation instance is launched by ASG after the rolling upgrade.

6.2 Early Detection of Error

Some errors, which take a long time before Asgard throws an exception, are detected by our mechanism earlier. Missing ELB (Elastic Load Balancer) is one of the faults causing such errors. Once an ASG is associated with an ELB, any new instance must be registered with the ELB to become "in service," i.e., receive traffic. If the ELB becomes unavailable, ASG cannot successfully launch new instances. Asgard's configuration includes a setting for the maximal number of retries in case of failed startup. Asgard will print an error log and abort the rolling upgrade operation after retrying the maximal of times.

To simulate the case of a failed ELB, we manually deleted the ELB associated with the ASG after one instance is successfully launched by Asgard. After around 15 minutes, Asgard printed the final error, shown as below, which does not provide useful information about the potential cause of the fault.

[2013-08-19 15:30:09,165] [Task:Pushing ami-17cf5d2d into group sdmcmmapp--firstASG for app sdmcmmapp] Exception:com.netflix.asgard.push. PushException: 1 of 4 instance relaunches done. Startup failed 6 times for one slot. Max 5 tries allowed. Aborting push.

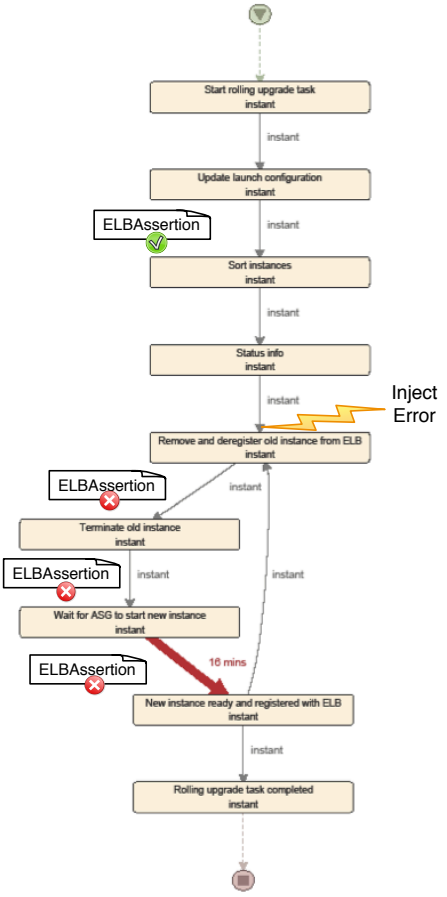

Figure 6. Allocation of ELB existence assertions.

To detect such errors, we added an assertion after every critical step to check if the ELB was still alive. As shown in Figure 6, 13 extra evaluation log lines are added in total. When the ELB is deleted after the first node is successfully launched, the evaluator associated with the assertion at the next step of the fault step (remove and deregister second instance from ELB) detected the missing ELB and printed a log line as below. "sdmcmmappELB" is the ID of the ELB associated with the ASG. The timestamp shows that our evaluation detected the missing ELB after approx. 2 minutes (roughly 13 minutes earlier than Asgard with the above configuration). The time difference depends on the maximal retries, so Asgard could report the error more quickly if configured accordingly.

[2013-08-19 15:17:59,205] [postcondition] [Task:Pushing ami-17cf5d2d into group sdmcmmapp--firstASG for app sdmcmmapp] [Step:pushstep2] ELB sdmcmmappELB does not exist.

6.3 Detection of Silent Error

There are some subtle configuration errors that escape Asgard completely, and result in an erroneous (not failed) system after the rolling upgrade. For example, a race condition during deployment can cause the launch configuration associated with the ASG to be changed during a rolling upgrade. Asgard doesn't detect such modifications of the ASG configuration. The printed logs are exactly the same as successful runs. However, the ASG ends up with 4 instances using two different launch configurations, for example, different AMIs, Kernel IDs, or security groups.

As we did for the case of a missing ELB, we added an assertion after every critical step to check the launch configuration used by the ASG. Thus, we added another 13 extra evaluation log lines into the original Asgard log. The following two log lines are produced by the evaluator: one line shows the configuration passing the check and the other shows it failing.

[2013-08-19 13:32:22,909] [postcondition] [Task:pushing sdmcmmapp----firstASG] [Step:pushstep3] sdmcmmapp--firstASG is using launch configuration sdmcmmapp --firstASG-20130819133202

[2013-08-19 13:38:09,425] [postcondition] [Task:pushingsdmcmmapp----firstASG] [Step:pushstep4] icse--firstASG is using launch configuration sdmcmmapp--firstASG-20130819124209 rather than sdmcmmapp--firstASG-20130819133202

6.4 Limitations of Using Logs as Triggers

Using only logs to trigger the evaluator has some limitations.

- For some operation steps duplicate log lines are produced by Asgard. For example, two almost identical logs lines are produced before a function is invoked and within the function respectively without indicating the context. In this case, the regular expression we use to match the log line must be very specific to distinguish the two different but very similar log lines to avoid triggering an evaluation twice.
- Some operation steps, especially the ones done by a third-party (like Amazon) do not produce any entries in the Asgard log. Thus, the operation steps provided by third-party services are invisible, unless Asgard has some built-in monitoring and logging mechanism to check what the third-party service does.

7. DISCUSSION AND FUTURE WORK

We have demonstrated that using an annotated process model to guide error detection is a feasible approach to detecting errors that occur during provisioning. Our prototype both detected errors faster than they would otherwise have been discovered and also detected errors that would not have been discovered until execution of the system being deployed.

We did this under favorable assumptions, however. The logs produced by Asgard are of high quality and come from a single source. We also implemented several steps of our proposed workflow manually. Our future work includes removing these favorable assumptions. We plan to:

1. Automate more steps. We hand crafted many of the intermediate steps in our error detection. Some of these could be, at least partially, automated. One candidate is to more tightly integrate the process mining and the post condition triggering.
2. Deal with other and multiple sources of logs. Asgard produces very high quality logs. We need to be able to construct a process model and trigger assertion evaluation when the logs come from different sources and are of lesser quality.
3. Diagnose errors. Detecting errors early provides more possibilities for error diagnosis and we plan to explore some of those possibilities.
4. Generalize our current approach for different contexts. We are investigating applying our process mining technique into VMware environments, applying our approach into the Chef framework, and working on decoupling the creation of the process models from logs.

8. ACKNOWLEDGMENTS

NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program.

9. REFERENCES

1. D. Oppenheimer, A. Ganapathi, and D. Patterson. Why do Internet services fail, and what can be done about it? The 1st USENIX Symposium on Internet Technologies and Systems, 2003.
2. Yang, J., et al. MODIST: Transparent Model Checking of Unmodified Distributed Systems. The 6th USENIX Symposium on Networked Systems Design and Implementation, April 2009.
3. Xu, X., Zhu, L, Bass, L. Lu, Q., Fu, M. Modeling and analyzing operation processes for dependability. Abstract, IEEE/IFIP International Conference on Dependable Systems (DSN), Budapest, Hungary, June 2013.
4. Gunawi, H.S. et al., FATE and DESTINI: a framework for cloud recovery testing," in proceedings 8th USENIX Symposium on Networked Systems Design and Implementation, March 2011.
5. Van der Aalst, W.M.P, Process Mining: Discovery, Conformance and Enhancement of Business Processes, Springer Verlag, 2011.
6. Lu, F., Fault Diagnosis Using Process Mining, MS Thesis, Eindhoven University of Technology, *alexandria.tue.nl/extra1/afstversl/wsk-i/fliu2011.pdf*
7. Wikipedia, Levenshtein distance http://en.wikipedia.org/wiki/Levenshtein_distance
8. Watson, B. W, A new algorithm for the construction of minimal acyclic DFS, Science of Computer Programming, Vol 48, Iss2-3, Aug, 2003.