The Journal of Systems and Software 110 (2015) 205–221

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Using simulation to evaluate error detection strategies: A case study of cloud-based deployment processes

Jie Chen[a,e,*], Xiwei Xu[b], Leon J. Osterweil[c], Liming Zhu[b,d], Yuriy Brun[c], Len Bass[b,d], Junchao Xiao[a,f], Mingshu Li[a,f], Qing Wang[a,f]

[a] Laboratory for Internet Software Technologies, Institute of Software, Chinese Academy of Sciences, Beijing, China
[b] NICTA (National ICT Australia), Australian Technology Park, Eveleigh, Australia
[c] College of Information and Computer Sciences, University of Massachusetts, Amherst, MA, USA
[d] School of Computer Science and Engineering, University of New South Wales, Sydney, Australia
[e] University of Chinese Academy of Sciences, Beijing, China
[f] State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China

ARTICLE INFO

Article history:
Received 2 December 2014
Revised 12 July 2015
Accepted 25 August 2015
Available online 6 September 2015

Keywords:
Process modeling
Simulation
Deployment process

ABSTRACT

The processes for deploying systems in cloud environments can be the basis for studying strategies for detecting and correcting errors committed during complex process execution. These cloud-based processes encompass diverse activities, and entail complex interactions between cloud infrastructure, application software, tools, and humans. Many of these processes, such as those for making release decisions during continuous deployment and troubleshooting in system upgrades, are highly error-prone. Unlike the typically well-tested deployed software systems, these deployment processes are usually neither well understood nor well tested. Errors that occur during such processes may require time-consuming troubleshooting, undoing and redoing steps, and problem fixing. Consequently, these processes should ideally be guided by strategies for detecting errors that consider trade-offs between efficiency and reliability. This paper presents a framework for systematically exploring such trade-offs. To evaluate the framework and illustrate our approach, we use two representative cloud deployment processes: a continuous deployment process and a rolling upgrade process. We augment an existing process modeling language to represent these processes and model errors that may occur during process execution. We use a process-aware discrete-event simulator to evaluate strategies and empirically validate simulation results by comparing them to experiences in a production environment. Our evaluation demonstrates that our approach supports the study of how error-handling strategies affect how much time is taken for task-completion and error-fixing.

1. Introduction

Understanding and evaluating complex real-world processes are made more difficult by the challenges in understanding how strategies for diagnosing and repairing errors affect the results produced by these processes. For example, domain experts intuitively know that human-executed process steps are relatively slower and more error-prone than automated steps, but might be more amenable to interactive error diagnosis and the prevention of overreaction by automated error recovery systems. Intuition also suggest that scripts can reduce errors and perform steps much faster than humans can, but that the use of scripts can propagate errors much more quickly and make error diagnosis more difficult. Similarly, common sense suggests that meticulously verifying the outcome of each step at a lower-level of granularity in an operational process helps to prevent downstream failures, but that doing so is expensive and can slow down the overall process. Informal guidelines are widely used in some application domains to decide which steps should be performed by humans and which by scripts, but such guidelines are often weakly justified. Such weakly justified guidelines are also often used to decide how frequently step outcomes should be verified and validated. Experts in other process domains similarly use other weakly justified guidelines.

These guidelines, and common knowledge should be justified, or replaced, by careful studies that provide clear, carefully reasoned justifications for specific approaches and practices. It is particularly important to provide justifications for practices relating to the

* Corresponding author at: Laboratory for Internet Software Technologies, Institute of Software, Chinese Academy of Sciences, Beijing, China. Tel.: +86 13466401225.
E-mail addresses: chenjie@itechs.iscas.ac.cn (J. Chen), Xiwei.Xu@nicta.com.au (X. Xu), ljo@cs.umass.edu (L.J. Osterweil), Liming.Zhu@nicta.com.au (L. Zhu), brun@cs.umass.edu (Y. Brun), Len.Bass@nicta.com.au (L. Bass), xiaojunchao@itechs.iscas.ac.cn (J. Xiao), mingshu@itechs.iscas.ac.cn (M. Li), wq@itechs.iscas.ac.cn (Q. Wang).

http://dx.doi.org/10.1016/j.jss.2015.08.043
0164-1212/

detection and correction of errors in complex processes. This paper presents one approach for providing justifications to such guidelines and knowledge. This approach uses error-seeding and discrete-event simulation to evaluate the effectiveness of error-detection and correction strategies. We illustrate and evaluate our approach using real-world error-prone processes employed in the domain of cloud computing.

Cloud computing processes (such as those related to the deployment, upgrade, failover, and reconfiguration of cloud-based applications) are particularly appropriate as evaluation vehicles because they are complex and error-prone. These processes are often orchestrations of intricate interactions among cloud infrastructure entities, application software, tools, and human activities. This complexity and the reliance on humans to make key decisions in time-critical situations make these processes particularly error-prone. Moreover, pressures for evolution of the underlying applications, and the emergence of continuous deployment practices are resulting in the need to exercise these processes as frequently as tens of times a day. The challenges of orchestrating these interactions at such high frequency and under uncertainties that are inherent to cloud environments can be considerable, increasing still further the propensity of such processes to error (Zhu et al., 2015). Errors can necessitate additional operations such as time-consuming troubleshooting, undoing steps, and problem fixing and redoing the undone steps. These operations can be expensive and error-prone themselves. To deal with this propensity for errors, these processes typically incorporate strategies for detecting, diagnosing, recovering from, and preventing errors. But the performance characteristics of these strategies can be subtle and hard to fully understand. For example, automated error detection and tolerance may reduce error rates and detect errors earlier but can also mask accumulative subtle errors leading to major outages and making error diagnosis more difficult. And fully automated *overreaction* to an initial small error is often the cause of major failures (Yuan et al., 2014). Humans, on the other hand, may spot errors that computers may miss, but are often slow, which can impede system progress. Therefore strategies for synthesizing these two approaches should be considered carefully as they yield trade-offs between efficiency and reliability. Choosing the wrong strategy may result in wasted resources or errors that propagate unnoticed. There is a need for research on approaches for effectively deciding on appropriate and effective strategies.

These challenges become particularly acute when dealing with large-scale applications that run in distributed cloud environments. The execution platform for a modern large-scale cloud-based system might consist of thousands of nodes, and the maintenance of such a system may require dozens of changes (e.g., the incorporation of new versions of software utilities) a day (Etsy, 2013), each of which may require the execution of a complex process of collaboration between automated tools and a busy operations team. As a consequence, errors are frequent. According to Gartner, "Through 2015, 80% of outages impacting mission-critical services will be caused by people and process issues" (Colville et al., 2010). Some errors arise from faulty system executions, which then trigger operator reactions, such as executing a complex remediation process that itself might be flawed. Errors of this kind, and their cascading effects, may have a significant impact on overall operational costs.

The deployment of cloud-based applications relies on the smooth performance of such key processes as preparing the environment, loading pre-baked virtual machine images into the environment, applying and propagating the necessary configurations, activating and deactivating the new and old versions, conducting small-scale canary testing, and rolling execution images out to perhaps thousands of nodes. All of these processes are complex, error-prone collaborations between human operators and automated systems. They are difficult to test using traditional testing approaches, despite attempts to treat them like regular applications by the Infrastructure-as-Code movement. It is important that errors be identified and handled within minutes or seconds, especially in the case of high-speed, high-capacity systems in domains such as finance, healthcare, and transportation, all critical components of key societal infrastructure. Although humans and automated tools collaborate in performing these processes and in dealing with errors, strategies for this collaboration are presently only guessed at.

We address these challenges by creating a general framework for evaluating error-detection and correction strategies, and then applying the framework to the domain of cloud computing. Specifically we

- *created a framework to integrate approaches for error detection and repair into complex processes. These approaches have characteristics that are demonstrably superior to current best practices, which are often simply weakly justified guidelines based on anecdotal observation and experience.*

And then, we

- *used this framework for sample complex cloud-computing processes, which, unlike the (presumably) well-tested software systems whose deployment they manage, are neither well understood nor well tested.*

In this paper, we model some example deployment operations as processes, each consisting of a collection of steps. Each step is executed by an agent, who is either an automated script, an assistive tool, or a human. Each step requires different amounts of time and various resources, such as computing power, a readied environment, or cloud computing nodes. We focus on two complex and representative processes, deployment and rolling upgrade, to illustrate our approach. We augment an existing process modeling language to define these error-prone processes, and use it to model precisely when errors can occur, the types and distributions of those errors, and the processes involved in checking for and correcting these errors. Recognizing that the error-checking and correcting processes themselves can both miss some errors and themselves actually create other errors, our framework supports modeling these situations as well.

We use a process-aware discrete-event simulator to show the overall effects that strategies can have on the final outcomes of process execution, and to suggest improvements to the processes. We carry out our simulations within a framework that incorporates detailed and precise models of these processes, populated with empirical data and measures obtained by observing real-world process executions. Our simulations are designed to represent realistic error-detection and repair scenarios that take place in the real world, and to support accurate comparisons of different approaches for dealing with these scenarios. For example, we use our framework to answer questions such as "How frequently should an error-checking action be performed, and at what level of granularity?" and "How much does increasing error-checking frequency reduce the risk of system failure?" Our view is that answers to such questions lead to cost-benefit trade-off analysis that could help developers and operators select policies that positively affect system operational costs and product quality.

Finally, we validate our framework and approach by comparing the models and results obtained from simulation studies to observed measurements of multiple large-scale executions of the processes in real cloud computing settings on the Amazon Web Service (AWS)[1] platform. Our results demonstrate that the simulations make reasonable predictions that can help actual operations personnel to conduct what-if analyses to support making better decisions. This, in turn, supports our view of the effectiveness of our overall framework and approach.

The rest of the paper is structured as follows. Section 2 introduces our modeling approach and Section 3 describes our simulation

[1] http://aws.amazon.com/.

approach. Section 4 evaluates our approaches using two case studies. Section 5 places our work in the context of related research, and Section 6 summarizes our contributions and future work.

2. Modeling approach

A key foundation of this work is the ability to define precisely and in detail the kinds of complex cloud computing deployment and maintenance processes described above. These processes are intricate collaborations between humans and software, where there is the clear possibility that either or both may perform incorrectly, requiring periodic checking and repair, both of which may vary in their effectiveness.

We modeled these processes using Little-JIL (Wise et al., 2006), a language having a number of features that proved to be particularly useful in supporting the specification of some difficult features of these processes. A particularly important feature is the ease with which Little-JIL can be used to specify how and where a process execution may deviate from the desired or expected, perhaps due to the commission of an error, or the occurrence of an adverse event in the execution environment. We characterize such a process execution as 'non-normative, in contrast to a 'normative' execution, namely one in which the performance of the process proceeds as expected and produces expected results. In some literatures such a normative performance is sometimes referred to as a 'happy path'. In contrast, a 'non-normative' performance is one in which something unusual, untoward, or undesired takes place. Often, as is the case in this paper, the non-normative performance is not desired, but as it may not be unexpected, appropriate reactions and responses can be defined so that they can be activated as needed to mitigate the negative effects of the non-normative execution.

In creating one of these models we began by capturing the normative, or desirable, behavior of the process, and then elaborated it to an error-prone behavior model, by specifying the possible errors, error responses and error repairs that could take place at every step. Our models of non-normative behavior included representing the types of error occurrences, the conditional probabilities of the different error types, and differences in error detection delay resulting from different detection approaches.

We now use the rolling upgrade process as the basis for examples of how some of the principal features of Little-JIL and our error-prone behavior model support the clear representation of the various kinds of error commission, detection, and repair to be described in more detail subsequently in this paper.

In these descriptions it will be important to distinguish clearly among the following four terms, which we will use often throughout the rest of this paper:

- Error: an incorrect action performed in a step, which might be attributable to incorrect performance by the performer of the step, an input to the step that is incorrect, or the occurrence of an event in the system environment that has an adverse effect on the step or its performer.
- Defect: an imperfection that exists in an artifact due to an error that has occurred during the performance of a step.
- Failure: execution of a step that causes the step to fail to accomplish its intended purpose.
- Exception: a non-normative outcome of the execution of a step.

2.1. Rolling upgrade process

Rolling upgrade is an important process for continuous deployment and high frequency releases. Assuming a current version of an application is running in the cloud and consists of a large collection of old version virtual machine (VM) instances, a rolling upgrade process then deploys a new version of the application, upgrading a small number of VMs at a time. Once this small number of VMs is deployed with the new version, an equivalent number of the old version VMs can be removed. Repeating this process many times will result in completing the deployment. Rolling upgrade is inexpensive compared to spinning up a very large collection of VMs containing the new version and then switching all of them with all of the nodes running the old version all at once. This incremental strategy also enables more careful monitoring of the rollout so a rollback can be done sooner if problems arise.

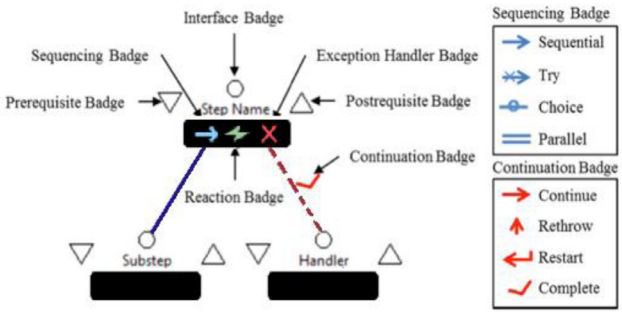

Fig. 1. A Little-JIL step.

In the Amazon Web Services (AWS) public cloud, one application is often organized into one or more Auto Scaling Groups (ASG) that consist of a set of VMs or nodes instantiated from an Amazon Machine Image (AMI). A given application could be deployed either in a single ASG, or in multiple finer-grained ASGs. To upgrade such an application, a small number of nodes within an ASG must be gracefully deleted and replaced by updated ones until all the nodes in all ASGs have been replaced. Close monitoring and checking of intermediate results and the final result is needed. Human operators can carry out the steps and the verification, but automated aids are available to assist much of it. On the other hand, both humans and these automated aids can commit errors. For example multiple upgrading processes may be taking place simultaneously on different parts of the system running the risk of creating mixed-version mismatches. This suggests the need for periodic checking and consequent repair as integral parts of an appropriately robust rolling upgrade process.

In this example we assume that the application to be updated is deployed across a number of ASGs, each of which contains a set of nodes. A specifiable number of the nodes within each ASG can be upgraded concurrently. This two-level hierarchy facilitates management of the upgrading of the application by increasing the potential for parallelization in upgrading (of sibling ASGs and nodes within an ASG). But because an error in the upgrading of any node may cause the failure of the upgrade of the entirety of its containing ASG, larger ASGs make the process more vulnerable to error, and require more work to repair. Thus deciding on the number of ASGs, the number of concurrent nodes, and the error checking strategies to be used, are decisions of considerable importance.

2.2. Process model

In this work we have used Little-JIL, a rigorously-defined graphical language in which processes are defined as hierarchical decompositions of steps into substeps. A step is essentially a procedure called by its parent, where argument artifacts are passed between parent and child. A step represents an activity to be done by an assigned agent. As shown schematically in Fig. 1, a step has a name placed above a black rectangular step bar, a badge that represents control flow among the step's sub-steps (connected to the step by edges emanating from the left of the step bar), an interface badge (representing the step's input/output artifacts, the agent that is to perform the step, and the resources the step requires), badges representing optional pre- and

post-requisites, an exception management badge representing facilities for optional scoped exception management, and a message handling badge representing exception handling steps, which are facilities for unscoped exception management. A step with no sub-steps is called a leaf step and represents an activity to be performed by an agent in any way the agent chooses.

A step's scoped exception-handling facilities deal with exceptions thrown by its descendants using exception handling subprocesses defined as substeps connected by edges emanating from a large X on the right side of the step's step bar. Exceptions can be thrown as part of the execution of any step, and also by a step's pre/post-condition. Exceptions are typed, and parameters can be thrown along with the exception. Four different modes of continuation after the handling of an exception can be specified. Exceptions have proved to be particularly useful in supporting the clear and precise representation of how the normative execution of a process might be departed from in order to deal with unusual or exceptional situations (Lerner et al., 2010). The execution of an exception handler may itself raise exceptions, providing expressive power that is often particularly effective in specifying particularly intricate process details.

We note that Little-JIL has been used to support the modeling of a large number of processes in a wide variety of domains, including software development, healthcare, elections, and dispute resolution. Little-JIL is a rigorously defined language, and thus the creation of these models is very much a programming activity, requiring discipline and appreciation of the advantages and challenges of dealing with rigor. But Little-JIL process models have a visual representation, which makes them relatively accessible to domain experts who may not be comfortable or competent with rigorous programming, but who may still be quite capable of understanding the modeling work that has been done by others. In short, the writing of Little-JIL process models is the domain of programmers, but reading of such models seems quite accessible to others. Anecdotally it has been noted that nurses, doctors, and election officials all have had success in reading, understanding, and critiquing Little-JIL models that had been developed by others. This has led to the iterative development and improvements of large numbers of complex process models. For particularly complex models, these iterations have sometimes continued for months to assure the accuracy and completeness of the models. But this amount of effort, and the possibility of these iterations has typically greatly increased confidence in the accuracy and completeness of the models. Little-JIL and its associated collection of editors, tools, and analyzers are available for download from the Laboratory for Advanced Software Engineering Research (LASER). [2]

We now present an example that shows a Little-JIL definition of a process for carrying out a rolling upgrade that incorporates various approaches to error detection, response, and repair. The process is shown in Fig. 2. We do not describe all details of this process in order to save space, and also because many of these details are of tangential importance to the focus of this paper. Thus we address only a few process features that exemplify our approach to modeling complex processes. In particular, we focus on the kinds of mechanisms we have used to model how defects can be detected, how this raises exceptions, how these exceptions can be handled, and how normative process execution can then be resumed.

2.2.1. Normative process model

We begin by addressing the normative process, namely the way process execution proceeds when all steps are performed as desired.

The process modeled in Fig. 2 represents the entire **Upgrade Process** (including both normative and non-normative executions), showing that it consists of the sequential execution (denoted by the right-arrow on the left of the **Upgrade Process** step bar) of four substeps, namely **prepare AMI** for the upgrade, and then doing the **Upgrade** of all the nodes in the distributed environment, then activating the upgraded environment (by executing the **Active** step), and then testing the result by performing **Test** on it.

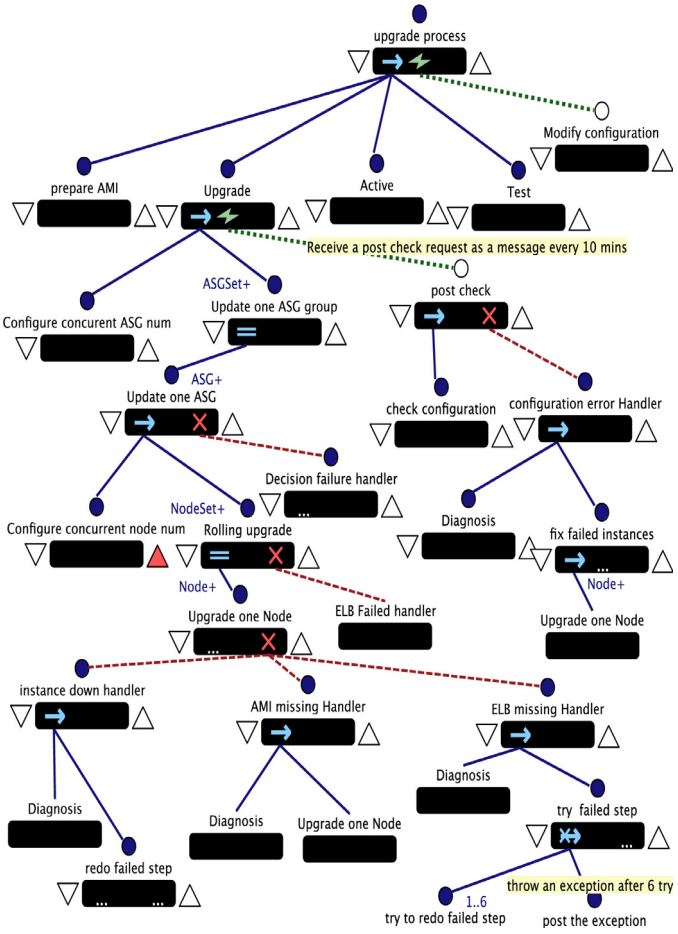

Fig. 2. Little-JIL model for an example rolling upgrade process.

Because of space limitations we elaborate here only the key step, **Upgrade**, which consists of the sequential execution of the **Configure concurrent ASG num** step**,** followed by the iterative execution of the **Update one ASG group** step. **Configure concurrent ASG num** decides how to decompose the upgrade into ASGs, with the decision embodied in the creation of the **ASGSet** artifact that the step generates as an output. **Update one ASG group** is then executed iteratively for each of the members of **ASGSet** (which is indicated by the **ASGSet+** annotation on the edge connecting **Update one ASG group** to its parent, **Upgrade**). **Update one ASG group** is defined, in turn, as the parallel (note the = sign on the left of its step bar) execution of **Update one ASG**, which is iterated **ASG+** times, where the elements of the **ASG** artifact were also computed by the execution of **Configure the concurrent ASG num.** Because this execution is done in parallel, the speed of the **Upgrade** process is increased. **Update one ASG** consists of the sequential execution of **Configure concurrent node num** (which defines the structure of the ASG), and **Rolling upgrade**, the parallel execution of **Upgrade one node.** The details of **Upgrade one node** are not shown here (denoted by the … ellipsis), and instead we focus on the representation of various approaches to error detection and repair.

2.2.2. Modeling the detection of defects

Real-world processes rarely proceed completely smoothly and any error operation will potentially introduce defects into the product

[2] Laboratory for Advanced Software Engineering Research (LASER) is in the College of Information and Computer Sciences, University of Massachusetts Amherst, USA web site, laser.cs.umass.edu.

being developed. As noted at the beginning of Section 2, we define a defect to be an imperfection in an artifact, perhaps created by the incorrect performance of a step. Defective artifacts will eventually be used as inputs to subsequently-executed steps, at which time the defect might be noticed. Alternatively, the defect might cause the creation of additional defects whose number and severity are likely to make the existence of the defects increasingly apparent, thereby triggering a search for the initial defect and a trace of its propagation. The sooner a defect is detected, the easier it is to detect and repair, thereby limiting the damage from the propagation of its effects. Defects can often be detected by executing steps that compare the evaluation of a step postcondition to the postcondition expected from a normative execution. Detection of a departure from the normative can cause an exception to be thrown. Here we describe three common strategies for interspersing such defect-detecting activities throughout the execution of a process. In this section, we show how the three common strategies for defect detection can be modeled as part of a process. The responses to such detection are discussed in the next section.

1. *Proactive defect detection*. Most fundamentally it is possible to represent the explicit proactive performance of defect detection in a process by inserting specific steps whose sole purpose is to determine whether the execution state of the process is as desired and expected from a normative execution, For example the **Test** step in Fig. 2 is to be performed after the entire upgrade process has concluded and the upgraded system has been in operation. **Test** will check the consistency of the upgraded system with normative behavior, including both current and previous performances and will alert operators or other systems (by throwing exceptions) if departure from the normative is found.

It often particularly important to identify defects produced by errors as soon as possible, in which case a more useful approach is to imbed these kinds of testing activities at close regular intervals throughout the process. Typically this is done by making these intermediate testing activities step post-requisites (or pre-requisites). For example as in the **Configure concurrent node num** step, the post-requisite can be a comparison between the parameter representing concurrent_node_num and some pre-defined maximum. This approach can be particularly effective by coupling a specific kind of checking with the results produced by a specific corresponding step.

2. *Reactive defect detection*. Another possible approach to defect detection is to cause exceptions to be thrown whenever the execution of a step fails, and cannot be finished normatively, for example due to lack of resources, lack of needed artifacts, timeouts, etc.

Typically this is done by causing these failures to be noticed, for example by looking for, and then propagating upwards, exceptions thrown by cloud infrastructure or system-level software. For example, note that if the **Upgrade one Node** step (whose substeps are not shown here) attempts to acquire and use a node (actually the execution of the leaf substeps of the **Upgrade one Node** makes this attempt) that is down, this will cause some kind of system fault. By putting in place handlers for this system fault, the process can then trigger its own higher-level exception whose handler can then take action to assure that the lack of this single node does not cause wider problems.

3. *Monitor-based defect detection*. Fig. 2 also shows how exception handlers can be triggered by external events or notifications, in addition to events arising from executing process steps or process-defined detection operations.

Thus, for example, the **Post check** step is carried out synchronously every ten minutes, driven by a timer-generated event. This step monitors the status of the process periodically by checking execution logs. This handler is attached to the middle of the **Upgrade** step bar, indicating that it is responding to an event that may originate anywhere in the process, or indeed from outside of the process itself (this is an example of an unscoped handler).

2.2.3. Modeling the handling of detected defects

Error analysis and repair activities follow the above defect detection activities, most often being carried out as exception handlers that are attached to an ancestor of the step that threw the exception (this is an example of scoped exception handling). Little-JIL exceptions are typed and handlers are defined to handle only exceptions of a single type. Thus, for example, **Configuration error handler** is in place to handle Configuration error exceptions, in this case thrown by the **Check Configuration** step.

In this paper, generally, an exception handling subprocess is modeled as consisting of two parts:

- The diagnosis part that is used to diagnose the cause of the exception and decide on a handling strategy for it.
- The exception handling part that is used to respond to the exception by carrying out the identified handling strategy.

There are a number of ways in which exception handlers can be defined, representing different approaches to diagnosis and repair of errors. This section describes different error handling structures used in this example. The behaviors of these handlers will be described in the next section.

The **Upgrade one node** step has three exception handlers, each for a different type of exception, each resulting from detection of a different kind of defect that may be detected in the course of executing **Upgrade one node**. Here, in this example, each of the three exception handlers for **Upgrade one node** takes a different approach.

Specifically, the **Instance down handler** is triggered when a node needed by the upgrading process is determined to be down. To deal with this exception, the handler first executes the **Diagnosis** step to decide which node is down, and therefore which step instance will need to be redone using a different node, which is presumed to be up and available. This step then communicates this information to the **Redo failed** step, which will repeat the exact step that failed using the different node.

In contrast, the **AMI missing handler** in Fig. 2, responds to a timeout exception thrown when execution of the Little-JIL step requires a particular AMI that is missing or unavailable (due, for example, to misconfiguration, superseding releases, or storage problems). The **Diagnosis** step is performed first and is followed by a recursive reinvocation of **Upgrade one node**, its grandparent step. Note that this recursive reinvocation supports the possibility that the response to the detection of a defect may itself be erroneous, effectively modeling the possibility of a potentially limitless chain of recursive exception handling. Presumably a real-world version of this process should limit the depth of this recursion, executing a step that invokes human intervention when the specified recursive depth has been reached.

Finally, the **ELB missing handler** shown in Fig. 2, models how a process may attempt to show some flexibility in trying to fix a problem by itself. In our example, the handler contains a “try” step, which allows its agent to try alternative sub-steps in left to right order until one of them succeeds. When a substep succeeds, the try step is complete and the failure has been addressed. In Fig. 2 the handler will try to redo a failed attempt to access the ELB at most 6 times. Then, if access still cannot be obtained, the last step under the “try” step will throw an **ELB failed** exception, which will then be handled by the **ELB Failed handler**, found at the next higher step in the execution hierarchy.

2.3. Error-prone behavior model

Process modeling can provide support for the analysis and improvement of processes that coordinate multiple people and tools working together to carry out a task. Steps in a process specify activities to be performed in order to support the overall goals and objectives of the process. In Little-JIL, each step specification includes a

specification of the types of entities that are required as resources order to perform the step. One of these resources is distinguished as the step's agent, which is an autonomous entity that is expected to be an expert in some part or parts of the process. In the processes modeled in this work, the step agent might, for example, be a human operator, an automated deployment tool, or a script. In addition, each Little-JIL step may specify the need for input artifacts such as installation packages and configuration files.

To understand and analyze a process fully, it is important for a process model to include descriptions of the behavior of a process under all possible circumstances, both normative and non-normative. This behavior is modeled by defining how the performance of the step transforms its input data into output data through the actions of the step's agents with the support of the step's other resources. *In this work we are particularly focused on the fact that the performance of a step (by either a human or a non-human agent) may be erroneous, thereby introducing defects into the step's output artifacts, or by incorrectly either throwing, or failing to throw, exceptions.* Thus, for example, a common misspelling of a word in the configure file during the deployment process could well be characterized as a non-normative performance. Such non-normative behavior may occur either because the incorrect performance of a previous step in the process has created defective artifacts that have propagated to the current step, or because of errors committed during the performance of the current step (e.g., because of the incorrect performance of the step's agent). The error may cause immediately identifiable problems (such as causing the incorrect throwing of an exception), or problems that may become manifest only later (perhaps even much later) in the execution of the process. Moreover, a single defect created by non-normative step execution may cause many exceptions to be thrown (perhaps even at different times), while, on the other hand the throwing of some exceptions may result from the erroneous performance of any one of many different steps.

To facilitate our work we designed a characterization of some principal kinds of behaviors, based on characterizations of the techniques that seem applicable to the effective diagnosis and repair of different kinds of error behaviors. We characterize step execution behavior as being:

1. *Normative* when everything goes as planned and desired, with all output data being correctly created and exceptions being thrown only when they are supposed to be thrown.
2. *Non-normative* when the step is executed erroneously and generates one or more output artifacts that are defective.
3. *Exceptional* when execution of the step results in throwing one or more *exception*s, indicating that some kind of defect has been detected (either correctly or incorrectly).
4. *Responsive* when the step is specifying a way that has been designed to address the need to handle a thrown exception.

The output artifacts generated by non-normative behavior may be the basis for throwing exceptions, thereby causing the execution of exceptional behavior and responsive behavior.

As noted above, an error in the performance of one step generally generates defective artifacts that may cause multiple exceptions to be thrown in the subsequent execution of the process. However, any given exception might be thrown by the erroneous performance of any one of potentially many different steps. Thus, for example, in the domain of management of cloud-based applications, when preparing an AMI, using the wrong version of Java one error may cause as many as three exceptions to be thrown (e.g., an inaccessible node exception, a system crash, or an incomplete component integration exception). On the other hand, throwing the corresponding node inaccessible exception may result from erroneous mapping of an IP address to a host name, or from any number of different version incompatibility errors.

We define an *impact* to be one (defect-detection, defect–repair) pair. We note that for each different *impact*, there may be many

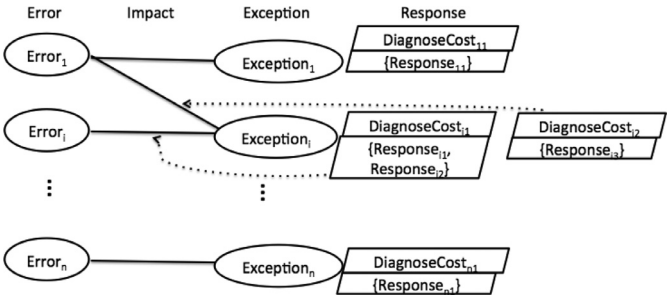

Fig. 3. Relationship among the defect, exception and response.

different approaches both to defect-detection and to defect–repair response. Each corresponding detection (exceptional behavior) and each repair (responsive behavior) may have a different profile of cost and effectiveness. Moreover, the cost and the effectiveness of both detection and of repair may vary depending on the contexts in which they are to be performed, where these contexts may incorporate such factors as the nature of the defective artifacts, the identity of step execution agents, and the proximity (in execution time) of the detection of the error to the time of its creation Fig. 3 depicts some examples of impacts and indicates their relationships to one another. We now formalize the specification of these different behavior models in the following sections.

2.3.1. Modeling normative and non-normative behaviors

The term behavior, as used in this paper, describes the way in which a step in a process transforms inputs into outputs. We define this more precisely as follows:

Let P be a set of parameters that are used as input artifacts to a step, S, and let $b_{A,S}$ be the behavior of the agent A performing step S. Then, if P′ is the set of output parameters created when agent A performs S, we write this as $b_{A,S}(P) = P'$.

Throughout this paper, unless otherwise noted, we assume normative behavior in describing the way a given step execution transforms input parameters into desired output parameters. These parameters are part of the artifacts that may be generated or passed through the step or may describe the status of an action (such as success or failure of an activity). Each input/output parameter in P/P' can be formally defined as a 2-tuple, < Name, Value > where:

- Name: is the specific name of the parameter.
- Value: is the value of the parameter, perhaps in the form of a list, a number, or a string.

In our example, the step **Configure concurrent ASG num** will take as input a set of nodes in the current configuration, and produce as output the number of concurrent ASG nodes.

Once the normative behavior has been modeled, it is important next to think about what might go wrong. In a process, the needed resources might be unavailable when they are needed, the actions that agents take might be incorrect or inappropriate, or deadlines might not be met. Steps carried out by automated scripts may also perform incorrectly due to errors in the scripts or miscalculations about the nature of the execution environment. In each of these cases, the abnormal operation will translate the input parameters into undesired ones, P'' (where P'' is usually not the same as P', the output set generated by normative execution).

Thus, to describe the non-normative behavior, we augment the definition of the normative behavior of a step by defining a set of errors that might be committed during execution of the step as part of the Little-JIL step definition. For each error we define a set of parameters that describe the conditions that must be met for the error to occur, the probability of this error, and the set of defects that will be injected into the process by this error. We assume that the probability of commission of this error will change over time at a learn rate,

which may be 0.0. The error definition contains a set of results that alter one or more of the output parameter values from the normative values. It is possible that every output parameter might be altered by any error. The output parameter values may be the basis for throwing exceptions in the Little-JIL process, thereby causing the execution of exception handling.

We define an error as follows:
Error = (Step, CPT_Err, Imps, LearnRate)

- Step: is the step in which the error may arise.
- CPT_Err: is the set of different conditions under which the error occurs, along with a specification of the frequency with which the defect is created under each condition. CPT_Err = $\{p_j \mid \text{Condition}_j\}$, where p_j is the probability that the error is created when the Condition_j is satisfied. The condition is a judgment statement with Boolean values used in determining whether the error should occur. For example, we can represent the fact that more experienced performers make errors less often by defining CPT_Err to be {(0.01, R.skillLevel > 3), (0.05, R.skillLevel ≤ 3)}.
- Imps: is a function that specifies the way output parameters are impacted by the error. The function takes an error condition and the desired output parameters as input and returns a specification of the defective values of the step output parameters that are created by the commission of the error. For example, if the rolling upgrade process upgrades 12 nodes concurrently, then **Configure concurrent ASG num** in Fig. 2 should output the value of the concurrent node number output parameter to be 12. To model an error in performing this step, Imps might specify that, under the conditions that are specified, **Configure concurrent ASG num** is specified to return some number other than 12.
- LearnRate: is a specification of the rate at which error occurrence changes with each execution of this step.

2.3.2. Modeling exception and response

It is of considerable importance not only to specify precisely the non-normative behaviors of a step, but also to provide a precise definition of how these non-normative behaviors could be handled if defects are detected. We specify the handling of the consequences of detecting non-normative behavior as exception handling. This requires, at the least, identifying the steps in which exceptions may possibly occur, identifying exactly what each exception is, identifying what the cause of the exception is, which steps and subprocesses are used to handle each exception, and then specifying how to proceed once each exception has been handled.

For the steps that may be performed incorrectly, a set of exceptions that might be raised by the execution of the step is defined as supplement to the step specification. The exceptions are defined similarly to errors, comprising a set of exceptional conditions that may arise, called the possible conditions, and the probability of the occurrence of each.

An exception is defined as follows:
Exception = (Step, Des, CPT_Exp)

- Step: is the step in which the exception may be thrown.
- Des: is a description of the exception (e.g., Node down, AMI missing).
- CPT_Exp: is the set of probabilities that the exception is thrown under different conditions, CPT_Exp = $\{(p_j, \text{Condition}_j)\}$, where p_j is the probability that the exception is thrown when the predicate Condition_j is true.

The exception may be thrown when the state created by the execution of the corresponding step satisfies the conditions in its description. When an exception event occurs it triggers a response by an exception handler that has been positioned in the Little-JIL process structure and designed to provide an appropriate response.

Although any exception may be triggered by more than one condition, any given exception can be trigged by only one condition at any given time. Therefore, our exception model defines the impact of each separate cause-effect pair, representing each of the different ways in which an exception can be triggered by one cause (under one condition). Accordingly, the specification of each different cause requires the specification of a different cost for diagnosis. But each cause may have more than one repair approach, each of which may cost a different amount of effort as described in Fig. 3.

We define an ExceptionResponse as follows:
ExceptionResponse = (Exception, Source, DiagnosisCost, Responses).

- Exception: is the definition of the exception.
- Source: is the set of errors for which a variable in Imps is also used to define a Condition in the CPT_Exp of an exception that is thrown as a result of the error:

$$error \in Source \Leftrightarrow \exists x\ (x \in error.Imps \wedge x \in CPT\_Exp)$$

- DiagnosisCost: quantifies the amount of effort required to find the cause of the exception. In our work, this is quantified as a time-valued function whose inputs represent the context in which the diagnosis is being made.
- Responses: is a set, $\{\text{response}_i\}$, each of which is a subprocess that can be used to recover from the effects of the occurrence of the error. It is used to guide the execution of the steps modeled in a specific subprocess defined as a handler of the exception. For example, the **redo failed step** is a choice step containing all the possible failed steps. After diagnosis, it will choose one specific step to be redone as the response to detected error.

2.4. Calibration of the process model

The process model we used as the basis of our work was initially defined and calibrated using observations and data from a real-world privacy analysis project (Li et al., 2013), a Log-based monitoring system, empirical study of Amazon Cloud API calls (Lu et al., 2013a) and the literature review (Nagaraja et al., 2004). For example, we took measurement data from observation of executions of processes for deploying Hadoop/HBase clusters in a real-world privacy analysis project, and we used these data to calibrate the error diagnosis and error repair times used in our deployment process simulation. We measured timing profiles of VM start time, monitoring related Amazon APIs for small scale rolling upgrades. We used them to calibrate execution timings and error detection timings for key steps in the rolling upgrade process. These calibrations were done before we ran simulations of large-scale executions of these cloud-based processes.

3. Simulation approach

Our work uses process-model-based discrete event simulation to evaluate different strategies for doing error diagnosis and repair in processes where fallible humans and execution of potentially defective scripts play key roles. Our approach combines the strengths of discrete event simulation, conditional probabilistic modeling, and error injection, exploiting detailed historical information about system executions that we derived from the observation of actual step executions and estimates of the effects of as-yet-undetected defects.

Our simulations were driven by a rigorously defined Little-JIL process definition and its extension as described in Section 2. These simulations take into account the possibility that some of the defined process steps will be performed incorrectly as human operators inevitably make mistakes, even when they are given explicit step-by-step directions. Errors vary in subtlety with some causing immediately-detectable problems and others creating problems that may be difficult or impossible to detect until much later in a process execution. Moreover, some errors may be easy to detect, while others may require considerable amounts of diagnostic effort. In addition,

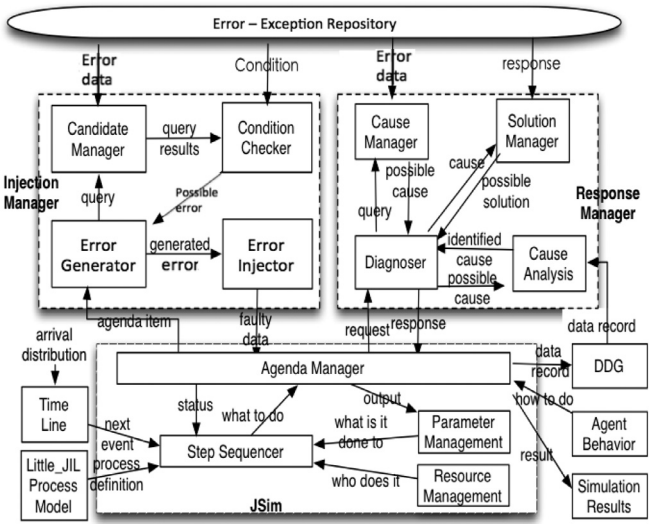

Fig. 4. System architecture.

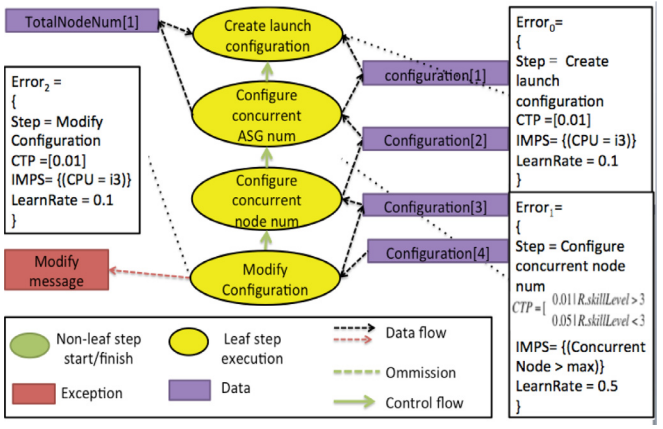

Fig. 5. An example of DDG. (For interpretation of the references to color in this figure, the reader is referred to the web version of this article).

some errors may cause multiple problems and defects, each of which may have a different profile of severity, detection difficulty, and repair effort.

To support the accurate simulation of these kinds of challenging execution scenarios, we have extended JSim (Raunak et al., 2011), an existing discrete-event simulator of Little-JIL process definitions, to allow for the possibility that some steps may be performed incorrectly, and that repair may occur at different places. The architecture of the system to support this is shown in Fig. 4, which we now describe in more detail in the following sections.

3.1. The JSim-based simulator

JSim is a previously developed system that simulates the execution of processes defined in Little-JIL. JSim builds on the capabilities of the Juliette process execution environment, which executes Little-JIL steps in the order dictated by Little-JIL semantics, assigning them to step execution agents as defined by the process and simulating the actions of the agents. Agent actions are simulated according to behavior specifications that are specified in the JSim resource manager described in detail in Raunak et al. (2013), which specifies how to transform input artifacts into desired output artifacts, how much time and effort are required to do so, and conditional probabilities that the agent will commit any number of specified errors. The results of a process simulation are recorded in the form of a DDG (Data Derivation Graph) (Lerner et al., 2011) that is used as the basis for determining the course of future process execution and agent behavior.

More specifically, JSim works as follows:

To begin, JSim initializes the simulation clock to zero and consults its event arrival stream for the first event to be simulated. JSim's Step Sequencer picks up each step to be simulated in the order specified by the process, and for each step consults the Resource Manager to obtain the needed agent and other resources and the Parameter Manager to obtain the step's input artifacts. The Step Sequencer packages all of this into an agenda item, which the Agenda Manager places on the agenda of the step's selected agent. The agenda manager monitors the status of all agenda items, detecting whether exceptions are thrown by a step execution, and whether any error has been injected by the system in the form of a defective output artifact. The duration of a step's execution is computed based on the agent's behavior specification, the difficulty of the step, and execution conditions, and is used to update the simulation clock.

While it is certainly necessary to specify in precise detail the normative execution of a process in order to support discrete-event simulation, our work requires going farther. In our work we are attempting to evaluate alternative approaches to error detection and correction. Therefore our work also requires the precise and detailed specification of how errors arise, creating defects, and how these errors and defects can be detected and repaired. Based on these specifications, different error detection and correction approaches can be compared.

For this project, we extended the JSim by augmenting it with the use of an Injection Manager and a Response Manager. In addition, an Error-Exception Repository was defined to store the descriptions of the errors, exceptions and the responses that may happen for each step in our models (as described in Section 2). The Injection Manager inspects the simulation status and, in collaboration with the Error-Exception Repository, decides whether or not to inject an error into the simulation of the step's execution. It injects these errors by adjusting the outcomes impacted by the generated errors (how these impacts are modeled is described in Section 2.3), thus causing faulty artifacts to be incorporated into the process simulation. Thus, the execution of a step may throw an exception either because of an incorrect result of a previous step, or because of some internal errors during the performance of a step. To handle an exception, a request is sent to the Response Manager component to identify all of the possible causes of the exception, all of the possible responses to each, and to decide what response is to be simulated.

Execution history information is stored in the Data Dependency Graph (DDG), which is generated automatically during process execution. In the simplified example DDG shown in Fig. 5 there are two types of nodes: ovals and rectangles. Ovals represent step execution instances and rectangles represent data instances. Red rectangles represent exception objects. Each instance is labeled with a name, and some of the names are also indexed to represent the different instances of the named entity instantiated at different points during process execution. There are two different kinds of edges in the diagram: data flow edges, which are used to connect data instances that are used and created to the step instances using and creating them, and control-flow edges, which represent the step execution sequencing. Fig. 5 also contains boxes that indicate possible errors and steps that may cause them. As an example, in Fig. 5, the step **Create Launch Configuration** initializes the *TotalNodeNum[1]* to 2 and *configuration[1]* to i5, which means that two nodes need upgrading with CPU i5. (In this example, we omit other related configuration context data such as memory size, software version, etc.). Then, **Configure concurrent ASG num** performs the actual upgrade with one ASG by modifying the parameter *configuration[2]* with concurrent ASG number 1. **Configure concurrent node num** takes *configuration[2]* as input and uses the value 2 for the concurrent node in the configuration

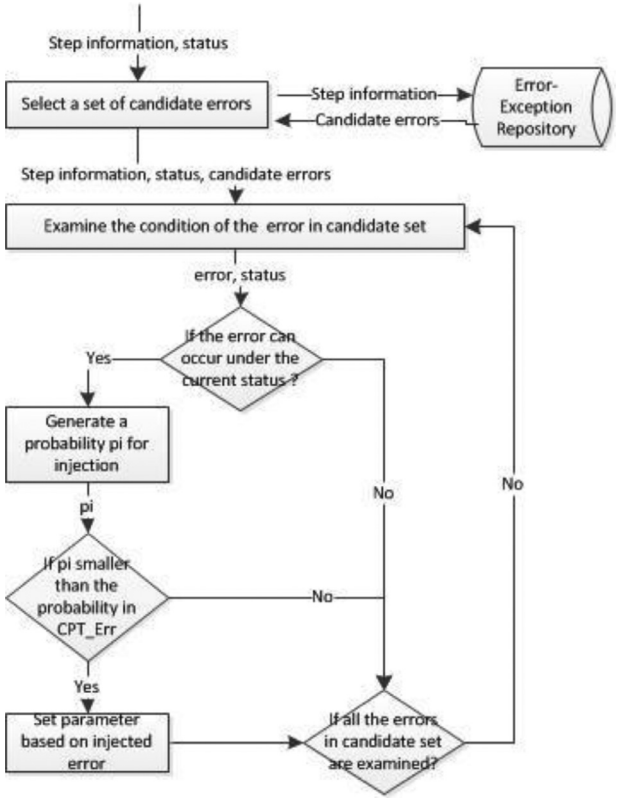

Fig. 6. Flowchart summary of the error injection process.

file marked as *configuration[3]*. This DDG might be generated, for example if a low-skilled operator erroneously tries to do a setup with concurrency larger than the allowed maximum (in which case we inject an error with probability 0.05—because the operator's skillLevel is less than 3). But before upgrade starts, another operator changes the configuration to CPU i3 for some reason. This modification will be triggered by an exception that causes the handler **Modify Configuration** to set the value of CPU to i3 in *configuration[4]*.

3.2. Injection manager

The actual injection of errors into a simulation is performed by an Injection Manager. In a normative (error-free) execution, the agent responsible for executing a step performs it as defined in the JSim Agent Behavior specification, creating the desired outcomes by transforming input artifacts into output artifacts. To simulate the creation of defects that require detection and repair, it is necessary to simulate non-normative execution of steps that inject errors in the form of defective outputs. These may, in turn, cause exceptional condition handling, depending on whether or not the generated defects are detected.

For every step that is simulated, the Injection Manager is invoked to decide whether an error is to be injected, and if so, what the error shall be, as shown in Fig. 6. For a given step simulation, the injection process starts with the Error Generator gathering step execution status information from the agenda of the agent performing the step, where the step's execution status is defined as:

StepStatus = (Step, Inputs, Outputs, Res); where

- Step is the step specified by the agenda item.
- Inputs is the set of artifacts received as input to the step, Inputs = $\{inp_1, inp_2, \ldots inp_n\}$.
- Outputs is a set of artifacts generated or propagated by the step. Outputs = $\{outp_1, outp_2, \ldots, outp_n\}$.
- Res is the set of resources assigned to perform the step.

Then, the Injection Manager consults the Candidate Manager to search the Error-Exception Repository to identify all possible ways in which the execution can be erroneous. This is done to determine a set of candidate errors that may occur in the execution of that step. For each candidate error the Condition Checker is invoked to filter out errors that are deemed to be of too low probability, based on current conditions as determined using current status and error information. The Condition Checker uses the condition part of the CPT_Err definition of each Error to make this determination. Thus, for example, the CPT_ Err specification may specify that an error can only be generated when the agent performing the step is a human. In that case, the Condition Checker will examine the type of the agent performing the step to determine if the error can be generated. Other execution status information, such as the phase of process execution, time limits that have been placed on step execution, or prior attempts to execute this step (perhaps by other agents) might also be used by the Condition Checker and compared to CPT_ Err specifications.

Once the Condition Checker determines that it is possible for an error to be generated, the Error Generator then evaluates the probability of the occurrence of each feasible error under current conditions and uses probabilistic distributions to determine which of the feasible errors to generate. It generates one or more errors comparing a randomly-generated number to the probability distribution for each error, under the current execution circumstances, as specified in CPT_ Err. Finally, the Error Injector will inject the error into the process by altering one or more parameters in its output artifacts accordingly, as specified by the Imps component of the defined Error.

3.3. Response manager

It is the job of the Response Manager to decide how to respond once an exception needs to be handled during the execution of a simulation. As noted above, defects caused by a non-normative process execution may be detected either by an agent in the process of executing a step or by an explicit checking step such as a post-requisite (as shown in Section 2.2.2). In either case the detection of the defect causes an exception to be thrown. Section 2.2.3 suggested a variety of overall approaches to handling such an exception. In this section we discuss this in more detail.

The response to a detected error depends on an analysis of the nature of the defect. This may be difficult because the defect may be detected long after it was created by a non-normative execution, during which time the effects of the error may have propagated widely. The Response Manager is designed to support careful analysis of symptoms to find the cause of the exception and select one possible solution. In our simulations the Response Manager is triggered explicitly by the execution of a **Diagnosis** step (e.g. see Fig. 2), which receives a request formatted as:

$$\text{Request} = (\text{Exception}, \text{FailedCondition})$$

where Exception is the exception instance thrown by the process and FailedCondition is the unsatisfied predicate that triggered the exception.

For each diagnosis request, the Response Manager is executed in three phases, as follows:

1. The Diagnoser uses the FailedCondtion to query for all possible causes. To find the cause, the Cause Manager consults the Error-Exception repository to obtain all errors that might have possibly caused the exception. An error will be selected as a possible cause if its Imps contains the defect and satisfies the FailedCondtion. Thus PossibleCause, the set of errors that might possibly cause the exception is:

$$error \in PossibleCause \Leftrightarrow \forall x\,(x \in FailedCondition \rightarrow x \in error.Imps)$$

For example, in Fig. 5, execution of the process may throw the **launch fail** exception if the CPU version in the configuration file is < i5. In that case, PossibleCause includes $error_0$ and $error_2$, which would alter the CPU configuration version to i3.

2. Next, the Cause Analysis component analyzes the related execution history data in the DDG to infer the possible causes of the exception. Cause Analysis traces backwards iteratively and recursively through the DDG to find the modification history for all of the parameters used in FailedCondition. For each reviewed step, Cause Analysis determines whether any of the parameters have been impacted by errors in PossibleCause. This trace back iteration stops when an error is identified as a cause. Cause Analysis identifies an error in SourceCause as the cause if the impact of the cause is reflected in the outputs of the step instances:

$$error \in \text{SourceCause} \Leftrightarrow \forall x \ (x \in error.Imps \rightarrow x \in step.Outputs)$$

As an example, the failed parameter in Fig. 5 is the configuration, which is changed in previous steps in the process. Cause Analysis backtracks through the process. First, **Upgrade one Node** will be reviewed and analysis will show that that step does not change the value of configuration. But review will show that execution of **Modify Configuration** may create $error_2$, which is one of the errors in PossibleCause. So, if the output configuration file has CPU = i3, then it will be identified as a source cause of the exception. If not, then checking will continue backwards through the DDG.

3. Finally, after the identification of a source cause, the Diagnoser uses Cause Analysis results to obtain a response from the Solution Manager. The Solution Manager queries the repository to find all possible ways to repair the error, based on the inferred cause (which may indeed be incorrect) and the defined ExceptionResponse model in the Error-Exception repository. At present each cause has only one response, which is then selected. In future work we will address the problem of how best to select the response that is most appropriate to the current process execution state. It should be noted that the repair actions themselves could be flawed requiring further fixing that will extend the process duration and workload. The execution time for the diagnosis step in the handler is modified taking into account the defined diagnosis cost. And the frequency of the error is updated using the LearnRate. After that, the process continues by seeking prior causes for the exception that has been repaired. In this way we address the need to repair ripple effects of defects whose detection was delayed, thus letting defects propagate.

4. Case study

This section describes initial evaluation of our approach through the simulation of two common deployment processes in a distributed cloud-based execution environment — rolling upgrade and continuous deployment. For both cases, we modeled the deployment process, possible errors, exceptions, and repair operations using our modeling approach. We then simulated both processes for large-scale operations and compared the simulation results and recommendations with the data we collected from corresponding process executions in real-world settings.

A Personal Computer with Intel Core CPU (2.9 GHz) and 8GB RAM was used to perform the simulations. Each simulation consisted of 500 executions of the Little-JIL model of the process. For each case study, 5 simulations were performed in order to ensure that 95% confidence intervals were attained for all performance measures.

Each execution of the Little-JIL model of the process consisted of a discrete event simulation performed using the JSim simulation engine. As described in the paper, each execution entailed the simulated performance of steps, some of which were error-prone, resulting in different executions of some steps, causing each execution of the process to potentially be different from any of the others. Thus, by running 500 executions we expected to cause a range of different execution behaviors. The decision to run 500 executions for each simulation was dictated by consideration of the running-time for each simulation. Then, to make sure the simulation results were sufficiently representative to support a thorough case study, we performed five simulation runs and checked to see that those five simulation runs produced sufficiently comparable results. We used statistical hypothesis testing to evaluate the hypothesis: the five data sets originate from the same distribution. We used the Kruskal–Wallis test to make sure these results from different simulations did not show significant differences at the $p = 0.05$ level. The mean p-Value for our case studies averaged 0.676, which means that our simulations produced representative results.

We populated the models of these processes with empirical data gathered from small-scale experiments and from the literature, and then used simulations of the processes to:

(1) Suggest the optimal number of concurrent groups and nodes to be processed in parallel, given various early error-checking strategies in the rolling upgrade process.
(2) Suggest the effectiveness of using humans, scripts, and prebaked images to perform various steps during the deployment process and identify those steps in these two processes that are the most sensitive, and therefore the most important to study.

4.1. Rolling upgrade simulation and validation

This section describes our simulations of a process for doing rolling upgrade of a distributed log monitoring application that consists of Redis,[3] Logstash, [4] ElasticSearch[5] and Kibana[6] running on the Ubuntu operating system. Each deployed instance of this process can be used to aggregate distributed logs produced by the customer's own applications. We have based our work on the study of 72 instances of this process, all deployed in AWS and all using Netflix's Asgard tool[7] to assist the rolling upgrade.

Of all the potential rolling upgrade process errors, one of the most challenging is the ASG mixed version error, which can be caused, for example, by changing a launch configuration during an ongoing upgrade. In executing this process, there are situations in which multiple operators may be allowed to manipulate the same ASG. Under such circumstances, the launch configuration used by an ASG could possibly be changed unexpectedly after the rolling upgrade starts. A changed launch configuration would not block the upgrade process, but could cause the ASG to end up with instances using two different launch configurations. A simple lockdown of the configuration by independent teams is not advisable in high frequency deployments because of resulting loss of efficiency and need for additional communication among teams. So it is important to find an optimal strategy for error-checking and repairing mixed version defects during the upgrade process.

4.1.1. Scenario 1: rolling upgrade without error

Before evaluating different error checking strategies, we did initial simulation and validation of the model of our process definition. In this experiment, the process is executed without any error but with different configurations.

[3] http://redis.io/.
[4] http://logstash.net/.
[5] http://www.elasticsearch.org/.
[6] http://kibana.org/.
[7] https://github.com/Netflix/asgard.

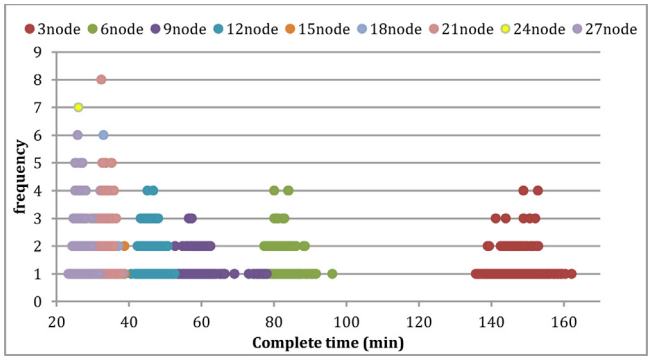

Fig. 7. Simulation: completion times with different concurrent node settings.

Table 1
Experiment: completion times of different rolling upgrade concurrent node settings (3 runs each).

6 nodes (min)		12 nodes (min)		18 nodes (min)	
Sim	Exp	Sim	Exp	Sim	Exp
Avg: 82.19	75.02	Avg: 46.36	40.23	Avg: 33.40	30.70
Min: 73.06	73.10	Min: 40.6	45.07	Min: 29.28	29.80
Max: 96.1	74.85	Max: 52.58	44.30	Max: 38.36	30.82

- Simulation

We ran the rolling upgrade simulation with different concurrent node settings. The results in Fig. 7 show that the upgrade completion times vary with the number of concurrent nodes (number of concurrent nodes with an interval of 3) in expected ways where small groups of concurrent nodes will lead to longer upgrade times. And the difference narrows with an increase in the number of concurrent nodes. Specifically, we found that the completion time has a larger range of variation when the number of concurrent nodes is smaller than 21.

- Validation

We then conducted upgrade experiments with three configurations that modeled our real 72-node deployment, including upgrading 6 nodes, then 12 nodes, then 18 nodes at a time. We chose only 6, 12, and 18 in the experiment because upgrading only 3 nodes in parallel would have be too slow and upgrading more than 18 in parallel violated our rule that not more than 25% of all nodes should be upgraded at any given time. The results in Table 1 compare the completion times observed in actual practice in a real-world setting with the average and median completion times of simulation (calculated from the data shown in Fig. 7).

The observed completion times and the simulation results are very close to each other (the difference in average time between simulation and experiments is 8.4% on an average). These results also show that the more concurrent nodes, the shorter the rolling upgrade process is and the 18-node concurrency indeed yields the fastest results as expected.

4.1.2. Scenario 2: rolling upgrade with errors

To further understand the impact of ASG configuration change during an upgrade, we ran our simulation while also injecting one configuration change fault into the process at a random point of time between 0 and 150 simulation minutes. All the instances launched after the launch configuration were then changed and were then using the wrong launch configuration and thus needed to be replaced to fix this error. The resulting mixed version defect does not cause any immediately observed failure and thus does not cause any immediate exception. Thus, this defect cannot be detected until the end of the process through testing, as is represented by the **Test** step in the process shown in Fig. 2. Once the defect has been detected, all of the nodes with the wrong configuration can then be replaced by the correct versions.

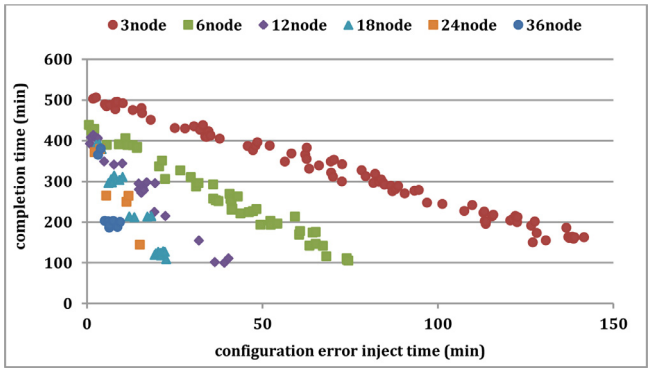

Fig. 8. Simulation: completion times with configuration change faults injected at different times.

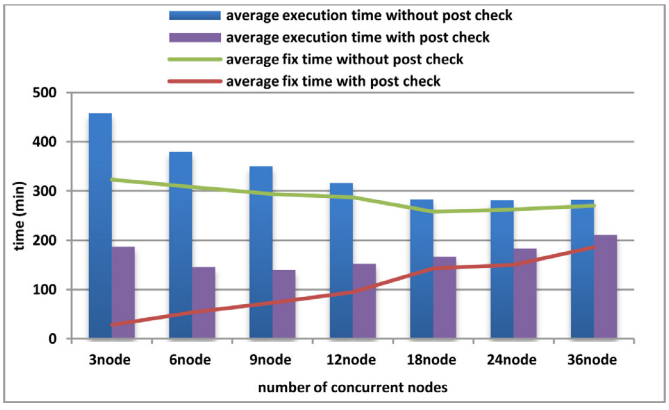

Fig. 9. Simulation: impact of the execution time with post check.

The results of simulating this error scenario are shown in Fig. 8. The results give a good picture of the completion time impact of the defect under different conditions. Generally, the smaller the number of concurrent nodes the longer the amount of time to complete the process. However, the relationship between the completion time and error injection time is different for different configurations. The process with less concurrent nodes is less sensitive to the error injection time, which is shown by the smaller slope of the graph of timing results shown in Fig. 8. And, conversely, the larger the number of concurrent nodes the more sensitive is the overall execution time of the process to the injection time, as the defect resulting from this error propagates more rapidly under these circumstances.

Various early error-checking strategies can be used in this process. Therefore, we then used simulation to explore two such strategies. As shown in the previous simulation, different error injection times will impact the completion times of processes having the same configuration. Thus, in this study, we injected an error within the first 10 min of the simulated process execution to limit the impact of the error injection time.

(1) Periodic checking

The first strategy introduced periodic error checking during process execution. To be able to detect changing the launch configuration earlier, we introduced an evaluator, which periodically checked the launch configuration used by ASG. In our experiments we checked this every 10 min. Once the defect was detected, we associated the ASG with the correct launch configuration. After the upgrade was completed, we terminated the instances with the wrong launch configuration, and created the same number of new instances with the correct launch configuration.

- Simulation

The simulation results are presented in Fig. 9, which shows that periodical checking significantly reduces process completion time.

Table 2
Experiment: 1-ASG *without* early error checking.

	Detection (min)	Wrong node	Fix (min)	Completion (min)	
				Exp	Sim
6	74.10	66	293	367.10	Avg: 379.19 STD: 0.168
9	51.73	63	253.26	304.99	Avg: 350.03 STD: 0.138
12	39.73	60	245.5	285.23	Avg: 316.36 STD: 0.115
18	29.37	57	238.61	267.98	Avg: 282.70 STD: 0.08

Table 3
Experiment: 1-ASG *with* early error checking.

	Detection (min)	Wrong node	Fix (min)	Completion (min)	
				Exp	Sim
6	10.23	12	49.05	123.11	Avg: 137.01 STD: 0.385
9	10.19	17	69.18	120.91	Avg: 134.65 STD: 0.327
12	10.18	21	82.38	122.22	Avg: 147.63 STD: 0.336
18	10.21	30	120.5	149.87	Avg: 163.34 STD: 0.275

However, with this strategy, the ASG with the largest number of concurrent nodes caused the largest number of nodes to have the wrong launch configuration, and required the longest time to fix the error. For larger numbers of concurrent nodes the differences between completion times with and without post checking are closer to each other, presumably because a group with a defect can be detected and repaired more quickly. Thus, doing post checking every 10 min did not detect faults until nearly the end of execution of the process. These results hold even when larger numbers of nodes are upgraded concurrently thereby causing more defective instances leading to longer repair times. Fig. 9 shows that 9-node concurrency leads to the fastest completion time.

- Validation

We executed these processes in our real-world distributed deployment environment to validate the simulation results. Tables 2 and 3 show the number of defective nodes at the time of detection, the repair time, and the total completion time respectively. The difference between the average completion times obtained from simulation and experiments is 9.6% on an average. These observation results confirm that the number of defective nodes depends on the number of concurrent nodes. The more concurrent nodes, the more correct new nodes in the ASG were launched within the previous detection interval, and the less time was spent doing repair. The result shows that upgrading 9 nodes concurrently resulted in the fastest completion time.

(2) Concurrent upgrade with fine-grained ASGs

We also used our approach to study the strategy of improving the performance of the rolling upgrade process by introducing more fine-grained ASGs rather using a single ASG, and allowing multiple ASGs to perform upgrades simultaneously. We conducted experiments with two configuration settings: 2 ASGs with 36 nodes in each, and 4 ASGs with 18 nodes in each. In this study, we injected the fault of changing the launch configuration for one of the ASGs. During rolling upgrade, the multiple ASGs were executed simultaneously. Error checking was done after each ASG was completed.

- Simulation

As above, we simulated the deployment of one, two and four ASGs with 3,6,9,12,15 and 18-node concurrency. Fig. 10 shows the average repair time after configuration faults were introduced. The simulation shows that under some circumstances, using multiple ASGs with less concurrency within each can outperform a single ASG with more concurrency. The experimental results show that both detection time and repair time are proportional to the number of concurrent nodes. Once the defect was detected, after the rolling upgrade completed, the defect was fixed by using the correct launch configuration to do an extra rolling upgrade on the faulty ASG.

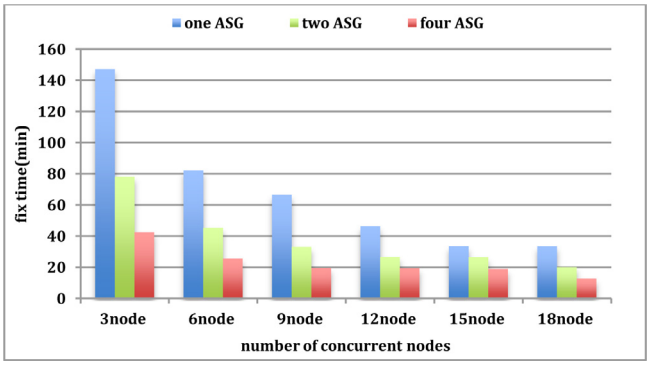

Fig. 10. Simulation: repair time with different numbers of ASG and concurrent node.

Table 4
Experiment: multi-ASG with configuration change fault.

		2-ASG, 36 nodes each	4-ASG, 18 nodes each
3	**Exp**	76.15	35.98
	Sim	Avg: 78.12 STD: 0.303	Avg: 42.5 STD: 0.189
6	**Exp**	36.63	20.62
	Sim	Avg: 43.04 STD: 0.212	Avg: 25.57 STD: 0.127
9	**Exp**	27.72	19.55
	Sim	Avg: 33.04 STD: 0.137	Avg: 19.49 STD: 0.09

- Validation

The actual results observed in our real-world deployment environment again validated these simulation results. The results in Table 4 are consistent with simulation results where the difference in average fixing time between simulation and experiments is 11.3% on an average. The results indicate that using multiple ASGs has the same effect on reducing overall execution time as early error checking with a single ASG.

4.2. Continuous deployment simulation and validation

Our second case study explored different continuous deployment processes for a Hadoop/Hbase cluster used in a privacy analysis project. We first modeled the deployment of a Hadoop/HBase cluster as a set of processes where the agents for key process steps might be either humans, automated deployment scripts on top of a lightly-baked AMI, or a heavily-baked AMI.

Each of these three kinds of agents have some advantages and some disadvantages. The characteristics of each are described in Table 5. We note that:

Table 5
Characteristics of the three types of agents.

Need for checking results	Heavily-baked AMI Do not need	Automated script Need	Operator Need
Speed	Longest preparation time.	Less preparation time than pre-baked AMI, and runs faster than human operator	Runs the slowest.
Fault rate	No fault	Has much lower fault rate.	Highest fault rate.
Difficulty of error diagnosis	–	Hard to diagnose	Easier to diagnose.

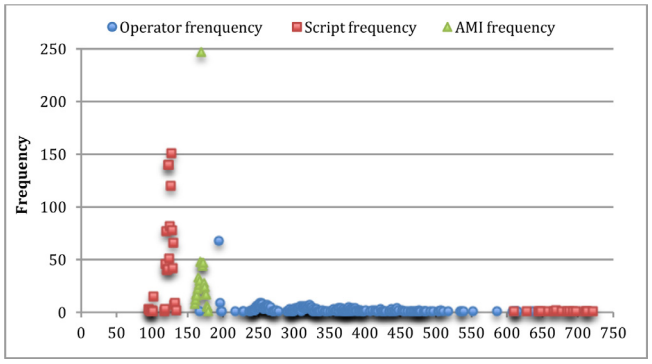

Fig. 11. Simulation: distribution of execution times.

(1) A human operator is relatively slower and inevitably makes human errors, such as typos and semantic mistakes (Keller et al., 2008) during this kind of deployment. However, manual installation is more amenable to interactive error diagnosis than installation performed by either of the other two kinds of agents.
(2) Automated script/deployment tools perform steps much faster than humans, and they avoid human errors. But the use of scripts may make error diagnosis difficult and the speed at which errors propagate complicates error detection and can necessitate more extensive repair actions.
(3) Deployment using heavily-baked AMI (so no more software is required to be installed by either humans or scripts after a VM is launched) is fastest of the three, and it is reasonable to assume that it is more reliable because it has presumably been carefully tested through extensive prior use. But preparing a heavily-baked AMI can be a time-consuming activity and implies terminating existing nodes for even small updates. The result typically offers far less flexibility than the other two approaches.

The first simulation was based on the basic description of the three resources. We ran simulations for the installation of a 4-node Hadoop/Hbase cluster using each of the three types of agent. Fig. 11 shows the distribution of completion times for the installations. On an average, the manual installation took longer than the other two, and in most cases, the script took the least time to complete. However, repairing errors introduced by a faulty script took longer than the time required to repair the errors introduced by the manual installation. Heavily-baked AMI execution time showed the least variation, and did not cause any installation failures.

We were also particularly interested in identifying bottleneck steps, because a bottleneck step in a multistage process can impede process performance most strongly. Generally, improving the performance of bottleneck steps can result in significantly higher throughput. So, besides the basic simulation, we also conducted sensitivity analysis to determine those actions whose improvement can be expected to have the greatest effect on completion time and overall failure rate. We adjusted the error rate of steps first by increasing it by 50% and then decreasing it by 50%, and recorded the percentage changes in execution time, fault and failure rates. As shown in

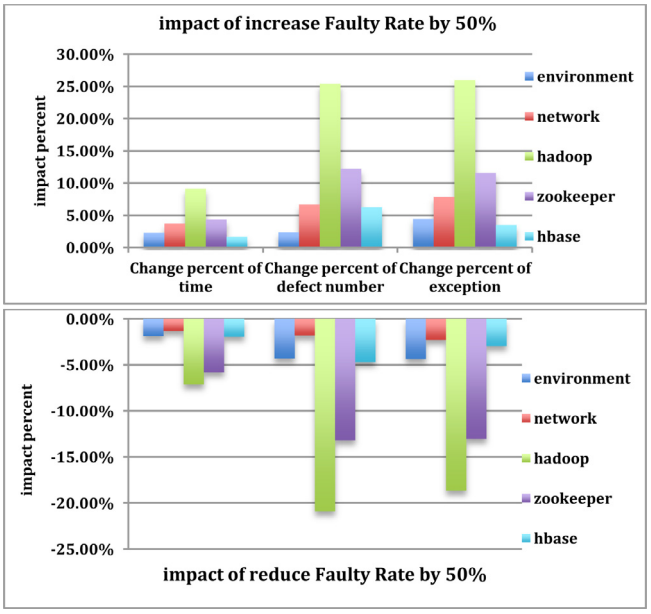

Fig. 12. Simulation: sensitivity analysis showing impact of fault rate on overall time, defect and exception.

Fig. 12, the installation of Hadoop is the most sensitive part of the deployment process. The installation of Hadoop requires more configuration than the other steps. Moreover, Zookeeper and HBase are installed on top of Hadoop. Thus, the correctness of Hadoop impacts both Zookeeper and HBase installation correctness.

We ran simulations to compare the results obtained using each of the three, and to demonstrate the tradeoffs among them under different configurations.

• Simulation

To illustrate the impact of cluster size on the deployment process, we simulated clusters with 2, 4, 10 and 100 nodes (execution of a 100 node cluster by a human operator was not done as the time required was considered to be excessive) and the results are as shown in Fig. 13. These results show that the completion time for manual installation largely depends on the total number of nodes because it is a sequential operation (on an average, 171 min for 2 nodes, 309 min for 4 nodes and 723 min for 10 nodes). The overall fault and failure rates for installations using a script are not affected much by the cluster size, because the same script is used to install all the nodes in the cluster. Thus, execution times increase slowly when the sizes of the clusters increase (on average, 123 min for 2 nodes, 135 min for 4 nodes, 199 min for 10 nodes and 1007 min for 100 nodes). The simulation results suggest that AMI is the most stable option for different sizes of clusters (on average, 169 min for 2 and 4 nodes, 176 min for 10 nodes and 268 min for 100 nodes). In general, the results also show that the use of the three different kinds of agents does not cause significant differences in the results obtained when there are only a small number of nodes (such as when there are only 2 nodes). And the difference increases as the number of nodes increases. The use of heavily-baked AMIs is advantageous only for large scale installations (such as when there are more than 100 nodes) where the long time

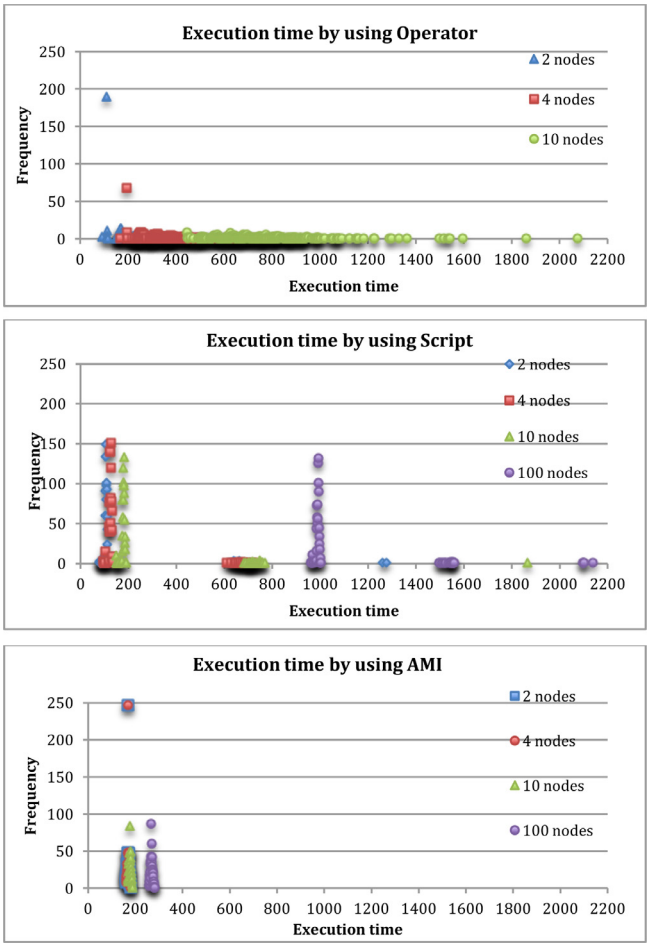

Fig. 13. Simulation: impact of different cluster sizes on execution times.

required to prepare the heavily-baked AMI is justified by its use on a large number of nodes.

• Validation

We then conducted experiments to compare different options for deploying a Hadoop Cluster. In our experiment, we use scripts and AMI respectively. We used Chef [8] to create our script. We deployed our own Chef server on AWS to host the cookbooks, the policies applied to nodes, and metadata about the node being manipulated by the chef-client. For pre-baked AMI, we used the Elastic Map Reduce (EMR) service provided by AWS. EMR helps deploy Hadoop on EC2 instances, and uses Amazon Simple Store Service (Amazon S3) to store input and output data.

We collected data for the deployment of a Hadoop cluster with two different settings, namely, 1 master plus 9 slaves and 1 master plus 99 slaves. The clusters were based on EC2 small instances with the Ubuntu operating system. For each cluster, we used both a script and AMI 3 times to do the deployment. As shown in Table 6, the script deployment completion time depends on the size of the cluster (The results in Table 6 do not include the time used for preparing the script/AMI).

4.3. Threats to validity

The main goal of this research is to present an innovative approach for gaining understandings of how error checking and repair should be done for complex human–computer processes such as continuous deployment and rolling upgrade processes, where humans and scripts often have to interact and neither has a definitive edge over the other. Although quantitative results have been obtained and presented, it is the overall approach, rather than these quantitative results, that is the main contribution of this research. Indeed the nature of our evaluations does seem to suggest that our approach can indeed be effective in gaining understandings, but also suggests that there is room to questions the validity of the specific quantitative results.

Table 6
Experiment: completion time of Hadoop deployment.

Script (min)		AMI (min)	
10 nodes	100 nodes	10 nodes	100 nodes
26	288	4.43	6.2
27	287	4.48	5.93
27	286	4.85	6.15

In particular, we note that we evaluated our approach only on two case studies, continuous deployment and rolling upgrade. We chose these two processes because they are both complex and representative. They are also relatively highly automated, and can be sufficiently well-defined to be the subjects of analysis. Moreover, that fact that both of them can entail contributions of both human and various non-human agents, they presented an opportunity to explore the relative merits of each. In addition, there remains some question about the accuracy and completeness of the various parameters that we used to characterize the performance and error-behaviors for each type of agent. There is also some doubt that we carried out a sufficient number of simulations to reduce the variability of the results that we have obtained.

On the other hand, our approach does appear to be promising as a way to support analysis of error detection approaches for other classes of processes. Our approach applies to processes where the performance of each individual step may be erroneous, and seems suitable for studying appropriate strategies for identifying these errors early and addressing the remediation of their effects. We believe that this approach should be applicable to any processes where errors can perturb the normal/desired action of a step and create output artifacts that can propagate that error more broadly. Our approach requires a specification of the structure of the process as a collection of steps, a specification of the behaviors of the agents performing the steps, a specification of how certain errors can occur during the execution of the steps, and how the defects created by these errors can be detected and addressed. Our approach seems to have limited applicability to processes for which the errors and defects cannot be described clearly or the process cannot be described precisely.

It should also be noted that the validity of our quantitative results also depends on how precisely, and to what level of detail, we have modeled the processes in the two example domains. While we have validated these processes through interviews with domain experts, the possibility of errors in the process models themselves remains, and could potentially reduce the accuracy and validity of our quantitative results.

Finally, although the experience data we used were taken from real-world systems and real-world projects, these projects were not carried out as a part of the regular daily operation of production systems. Thus, for example, there is a certain amount of concurrency in the actual rolling upgrade process that was not represented completely accurately in our simulations. Future simulations will need to be more accurate.

Consequently, we caution the reader to view our quantitative results with some skepticism, but do strongly encourage the reader to consider the value of the simulation and analysis framework that has been presented.

[8] http://www.opscode.com/.

5. Related work

Simulation models are generally employed as the basis for analyses of various kinds in various domains. In very early work authors such as Forrester et al. (1961) demonstrated the use of continuous simulation for specification and analysis of processes. The well-known System Dynamics framework for continuous simulation models is described in great detail by Madachy (2007). It has been used to model and analyze a number of engineering project management applications (Wolstenholme et al., 2003; Stavredes, 2001). Since the work of Abdel-Hamid et al. (1991), research and practice on software development have successfully adopted and continuously relied heavily on System Dynamics (Tvedt et al., 1996; Pfahl, 2001). More recently discrete event simulation (Fishman, 2001) seems to be a more prevalent approach. Authors such as Raffo (1996), Huo et al. (2006), and Al-Emran (Al-Emran et al., 2008) have applied this approach to the analysis and improvement of software development. Such models have also been particularly useful in supporting analysis of healthcare processes (Clarke et al., 2008; Lenz et al., 2013), web service processes (Estublier et al., 2005), and election processes (Raunak et al., 2006; Phan et al., 2012), as well.

But these earlier approaches have generally assumed that all process activities were carried out correctly, which undermines the realism of these models, and the relevance and accuracy of their results. Representing the possibility of incorrect performance and the propagation of its effects are fundamental problem for many process definition approaches. Thus, for example, models such as Markov Chains (Norris, 1998) and Petri Nets (Murata, 1989) have been used to specify and analyze processes. But the accurate representation of the ripple effects of incorrect performance and the use of conditional probabilities to accurately quantify these effects requires a proliferation of states in Markov Chains and places in Petri Nets. As this proliferation can be quite considerable, the use of these notations can be quite problematic.

Error injection is a well-known approach in the program testing community having been pioneered with the suggestion of Mutation Analysis (Budd et al., 1978), and followed by a considerable volume of work on various approaches to inserting errors into code and tracing their effects. But there has been a little work on using error injection to study processes. Little-JIL (Wise, 2006) is proving to provide a better basis for the precise and accurate representation of process activity errors and the propagation of their effects.

In cloud computing, analyzing operation-error troubleshooting from an artifact and provenance point of view can link issues back to source code, and improve configuration and log analysis (Xu et al., 2009; Tucek et al., 2007). However, these types of analyses do not view operations as processes that incorporate exception handling and are not aimed, as is our work, at improving these processes themselves. This previous work considered the processes as black boxes and analyzed their impact on overall availability using SRN (Stochastic Reward Nets) models (Lu et al., 2013b). Treating them as black-box actions and assuming that they perform correctly has severe limitations on the real world applicability of the analysis results. Fault injection frameworks, such as ConfErr (Keller et al., 2008), were used to investigate a system's reaction to various errors but they were not used to inject process-related operation errors. Other major work includes the use of simulation for cloud environment resource provision (Calheiros et al., 2011) but such work is difficult to be used for simulating operations processes such as upgrade.

6. Conclusions and future work

Human intensive processes, such as deployment processes entailing intricate interactions among cloud infrastructure, tools, and administrators are prone to errors. Various strategies for preventing the proliferation of these errors exist. But it can be difficult to decide when, where, and how to check for errors, diagnose them and redo certain steps in view of the fact that different strategies have different costs and efficacies. The structures of these processes and their artifact collections also have important impacts on the strategies to be used for detecting and correcting errors. The strategies can be complex and their implications are not immediately clear. Our framework facilitates experimentation with how different strategies affect process performance, reliability, and resource use.

In this paper, we extend the process-modeling language Little-JIL and provide a process-aware simulator. We describe the use of discrete event simulation to study the tradeoffs between different approaches to error detection and repair to configure and maintain the deployment process. The evaluation of our approach through discrete-event simulation was done for two common deployment processes in a distributed cloud-based environment—rolling upgrade and continuous deployment. These case studies suggested that our framework is reasonably easy to use and provides useful insights into the costs and benefits of the strategies we explored. Thus, we suggest that this framework can support analysts who are attempting to make decisions about trading off human effort for various types of automation, despite uncertainties introduced by the possibility of many types and distributions of possible errors.

We applied our approach only to the deployment process in distributed cloud-based environments. Thus, more work remains:

First, the approach should be applied to processes in other error-prone domains, to determine how well it can be generalized. This will require more empirical studies that will hopefully also validate our findings more firmly.

Second, the process oriented modeling and simulation approach should be extended to handle more types of errors that are likely to occur in this and other domains, and to extend the generality of our error and repair definitions. For example, we currently allow for only one repair for a single detected error, which is an inadequate reflection of real-world realities.

Third, we should complement the dynamic simulation approach taken in this work with complementary static analysis. In other work we have found that large, detailed processes such as those described here often have errors that compromise the validity of simulation results. Preliminary scans of our processes have not revealed such errors, but formal approaches such as model-checking of our process definitions are desirable to increase confidence in our results. In addition, we should apply Fault Tree Analysis, as was done in other work (Lenz et al., 2013) to identify how undesirable outcomes can result from erroneous performance of single steps, or pairs of steps. This would augment the simulation approach to identifying the most critical process steps, which we described in Section 3.

Acknowledgments

NICTA is funded by the Australian Government Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program. The research was also funded by the US National Science Foundation under Grants IIS-1239334, CNS-1258588, and IIS-0705772, by the Natural Science Foundation of China under Grants 91318301 and 91218302. Jie Chen performed this work as a visiting student at the University of Massachusetts Amherst, supported by China Scholarship Council.

References

Al-Emran, A., Pfahl, D., Ruhe, G., 2008. A method for replanning of software releases using discrete event simulation. Softw. Process: Improv. Pract. 13 (1), 19–33.

Abdel-Hamid, T., Madnick, S.E., 1991. Software Project Dynamics: An Integrated Approach. Prentice-Hall, Inc.

Budd, T.A., DeMillo, R.A, Lipton, R.J., Sayward, F.G., 1978. The design of a prototype mutation system for program testing. In: Proceedings of NCC, AFIPS Conference Record, pp. 623–627.

Calheiros, R.N., Ranjan, R., Beloglazov, A., De Rose, C.A., Buyya, R., 2011. CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. Softw.: Pract. Exp. 41 (1), 23–50.
Clarke, L.A., Avrunin, G.S., Osterweil, L.J., 2008. Using software engineering technology to improve the quality of medical processes. In: Proceedings of the 30th International Conference on Software Engineering. ACM, pp. 889–898.
Colville, R.J., & Spafford, G. (2010). Configuration Management for Virtual and Cloud Infrastructures. Gartner, http://www.rbiassets.com/getfile.ashx/42112626510.
Estublier, J., Sanlaville, S., 2005. Business processes and workflow coordination of web services. In: Proceedings of IEEE International Conference on e-Technology, e-Commerce and e-Service. IEEE, pp. 85–88.
Etsy, (2013) Infrastructure upgrades with Chef, http://codeascraft.com/2013/08/02/infrastructure-upgrades-with-chef/.
Fishman, G.S., 2001. Discrete-Event Simulation: Modeling, Programming, and Analysis. Springer Science & Business Media.
Forrester, J.W., 1961. Industrial Dynamics. The MIT Press, Cambridge, MA Reprinted by Pegasus Communications, Waltham, MA.
Huo, M., Zhang, H., Jeffery, R., 2006. A systematic approach to process enactment analysis as input to software process improvement or tailoring. In: Proceedings of the 13th Asia Pacific Software Engineering Conference. APSEC. IEEE, pp. 401–410.
Keller, L., Upadhyaya, P., Candea, G., 2008. ConfErr: a tool for assessing resilience to human configuration errors. In: Proceedings of IEEE International Conference on Dependable Systems & Networks with Ftcs & Dcc, pp. 157–166.
Lenz, D.R.R., Miksch, S., Peleg, M., Reichert, M., Riano, D., ten Teije, A., 2013. Process support and knowledge representation in health care. In: Proceedings of BPM Joint Workshop. Tallinn, Estonia ProHealth 2012/KR4HC.
Lerner, B., Christov, S., Osterweil, L., Bendraou, R., Kannengiesser, U., Wise, A., 2010. Exception handling patterns for process modeling. IEEE Trans. Softw. Eng. 36 (2), 162–183.
Lerner, B., Boose, E., Osterweil, L.J., Ellison, A., Clarke, L., 2011. Provenance and quality control in sensor networks. In: Proceedings of the Environmental Information Management Conference. EIM.
Li, J.Z., He, S., Zhu, L., Xu, X., Fu, M., Bass, L., … Tran, A.B., 2013. Challenges to error diagnosis in hadoop ecosystems. In: Proceedings of Large Installation System Administration Conference. LISA. ACM, pp. 145–154.
Lu, Q., Zhu, L., Bass, L., Xu, X., Li, Z., Wada, H., 2013a. Cloud API issues: an empirical study and impact. In: Proceedings of the 9th International ACM Sigsoft Conference on Quality of Software Architectures. ACM, pp. 23–32.
Lu, Q., Xu, X., Zhu, L., Bass, L., Li, Z., Sakr, S., Liu, A., 2013b. Incorporating uncertainty into in-cloud application deployment decisions for availability. In: Proceedings of the IEEE Sixth International Conference on Cloud Computing. CLOUD, pp. 454–461. IEEE.
Madachy, R.J., 2007. Software Process Dynamics. John Wiley & Sons.
Murata, T., 1989. Petri nets: properties, analysis and applications. Proc. IEEE 77 (4), 541–580.
Nagaraja, K., Oliveira, F., Bianchini, R., Martin, R.P., & Nguyen, T.D. (2004). Understanding and dealing with operator mistakes in internet services. In: Proceedings of Operating Systems Design and Implementation Symposium. OSDI, vol. 4, pp. 61–76.
Norris, J.R., 1998. Markov Chains. Cambridge University Press (No. 2).
Pfahl, D. (2001). An Integrated Approach to Simulation Based Learning in Support of Strategic and Project Management in Software Organisations. Fraunhofer-IRB-Verlag.
Phan, H., Avrunin, G., Bishop, M., Clarke, L.A., Osterweil, L.J., 2012. A systematic process-model-based approach for synthesizing attacks and evaluating them. In: Proceedings of the 2012 USENIX/ACCURATE Electronic Voting Technology Workshop.
Raffo, D., 1996. Modeling software processes quantitatively and assessing the impact of potential process changes of process performance. Manufacturing and Operations Systems, Doctoral dissertation, (Ph. D.) thesis. Carnegie Mellon University.
Raunak, M.S., Chen, B., Elssamadisy, A., Clarke, L.A., Osterweil, L.J., 2006. Definition and analysis of election processes. In: Software Process Change. Springer, Berlin Heidelberg, pp. 178–185.
Raunak, M.S., Osterweil, L.J., Wise, A., 2011. Developing discrete event simulations from rigorous process definitions. In: Proceedings of the 2011 Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S. Society for Computer Simulation International, pp. 117–124.
Raunak, M.S., Osterweil, L.J., 2013. Resource management for complex, dynamic environments. IEEE Trans. Softw. Eng. 39 (3), 384–402.
Stavredes, T., 2001. A system dynamics evaluation model and methodology for instructional technology support. Comput. Hum. Behav. 17 (4), 409–419.
Tucek, J., Lu, S., Huang, C., Xanthos, S., & Zhou, Y. (2007). Triage: diagnosing production run failures at the user's site. In: Proceedings of ACM SIGOPS Symposium on Operating Systems Review, ACM, vol. 41, no. 6, pp. 131–144.
Tvedt, J.D. (1996). An Extensible Model for Evaluating the Impact of Process Improvements on Software Development Cycle Time. Ph.D. Dissertation. Arizona State University.
Wise, A. (2006). Little-JIL 1.5 language report. Technical Report, University of Massachusetts, Amherst, 98-24.
Wolstenholme, E.F., 2003. The use of system dynamics as a tool for intermediate level technology evaluation: three case studies. J. Eng. Technol. Manag. 20 (3), 193–204.
Xu, W., Huang, L., Fox, A., Patterson, D., Jordan, M., 2009. Largescale system problem detection by mining console logs. In: Proceedings of Symposium on Operating Systems Principles. SOSP.
Yuan, D., Luo, Y., Zhuang, X., Rodrigues, G.R., Zhao, X., Zhang, Y., Stumm, M., 2014. Simple testing can prevent most critical failures: an analysis of production failures in distributed dataintensive systems. In: Proceedings of the 11th Symposium on Operating Systems Design and Implementation. OSDI.
Zhu, L., Xu, D., An, B.T., Xu, X., Bass, L., Weber, I., et al., 2015. Achieving reliable high-frequency releases in cloud environments. Software IEEE 32 (2), 73–80.

Jie Chen is currently working toward the graduate degree at the Lab of Internet Software Technologies, Institute of Software Chinese Academy of Sciences. She received the B.S. degree in software engineering from Xiamen University in 2009, also a B.S. degree (secondary one) in Economics from Department of Economics at the same University. Her research interests include resource scheduling, process analysis and simulation. She is currently doing research to refactoring practice in the process.

Xiwei Xu received her B.Eng. degree in Software Engineering from Nankai University (NKU), in Tianjin, China, in 2007. She received her Ph.D. degree from the School of Computer Science and Engineering at University of New South Wales (UNSW), in Sydney, Australia, in 2011. She is also with National ICT Australia (NICTA). Her research interests are in software architecture, business process and cloud computing.

Leon J. Osterweil is a Professor Emeritus in the College of Information and Computer Sciences at University of Massachusetts Amherst, where he had previously served as Dean of the College of Natural Sciences and Mathematics. He is a Fellow of the ACM. His paper suggesting the idea of process programming was recognized as the Most Influential Paper of the 9th International Conference on Software Engineering (ICSE 9), awarded as a 10-year retrospective. Prof. Osterweil was the recipient of the SIGSOFT Outstanding Research Award (2003), the SIGSOFT Influential Educator Award (2010), and the SIGSOFT Distinguished Service Award (2014). Prof. Osterweil has been the program chair and general chair of many Software Engineering conferences including serving as the General Chair of the 28th International Conference on Software Engineering (ICSE 28). He is a director of the International Software Process Association.

Liming Zhu is a research group leader and principal researcher at NICTA (National ICT Australia). He holds conjoint academic positions and teaches software architecture courses at both University of New South Wales (UNSW) and The University of Sydney. His research interests include software architecture, dependable systems and data analytics infrastructure. He has published more than 100 peer-reviewed papers. He formerly worked in several technology lead positions in software industry before completing a Ph.D. degree in Software Engineering at UNSW.

Yuriy Brun is an Assistant Professor in the College of Information and Computer Science at the University of Massachusetts, Amherst. He received the Ph.D. degree from the University of Southern California in 2008 and the M.Eng. degree from the Massachusetts Institute of Technology in 2003. He completed his postdoctoral work in 2012 at the University of Washington, as a CI Fellow. His research focuses on software engineering, distributed systems, and self-adaptation. He received an NSF CAREER award in 2015, a Microsoft Research Software Engineering Innovation Foundation Award in 2014, and an IEEE TCSC Young Achiever in Scalable Computing Award in 2013. He is a member of the IEEE, the ACM, and ACM SIGSOFT. More information is available on his homepage: http://www.cs.umass.edu/~brun/.

Len Bass has written two award winning books in software architecture as well as several other books and numerous papers in area of software engineering. His research interests include software architecture, software engineering, and DevOps.

Junchao Xiao is an Associate Professor in the Institute of Software at Chinese Academy of Sciences (ISCAS). He received his Ph.D. in Computer Science from ISCAS in 2007. He was a visiting scholar in the Department of Computer Science, University of Massachusetts Amherst from November 2008 to November 2009. His research interests include software process modeling, dynamic resource scheduling in complex processes and systems. He has published more than twenty papers in journals and conferences.

Mingshu Li is a Professor of the Institute of Software, Chinese Academy of Sciences. He received a Ph.D. degree from the Department of Computer Science, Harbin Institute of Technology in 1993; also a Master degree (secondary one) in Economics from Department of Social Sciences at the same university in 1995. He took his post-doctoral research in Institute of Software at the Chinese Academy of Sciences during 1993–1995 and Department of Artificial Intelligence at University of Edinburgh during 1995–1996. He is a Fellow of China Computer Federation (CCF). His major research interests are software process, software engineering methodology, requirements engineering, and operating syestems. He has published and co-authored more than one hundred research papers.

Qing Wang is a Professor of the Institute of Software, Chinese Academy of Sciences. She is the Director of the Lab for Internet Software Technologies. She has an intensive research and industry experience in the area of software process, including software process technologies and quality assurance and requirement engineering. She is a member of Cloud Computing Experts Association under Chinese Institute of Electronics and a Lead Appraiser of SEI CMMI SCAMPI. Qing Wang has led and is leading many important domestic and international cooperative projects. She has won various kinds of awards such as the National Award for Science and Technology Progress. She also is general co-chair of ESEIW 2015, PC member of some academy conferences and program co-chair of SPW/ProSim 2006, ICSP 2007 and ICSP 2008.