Challenges in Practicing High Frequency Releases in Cloud Environments

Liming Zhu[1,2,3], Donna Xu[1,3], Xiwei Xu[1], An Binh Tran[1,2], Ingo Weber[1,2], Len Bass[1,2]
[1]NICTA
[2]School of Computer Science and Engineering, University of New South Wales
[3]School of IT, Sydney University
Sydney, Australia
{firstname.lastname}@nicta.com.au

Abstract— **The continuous delivery trend is dramatically shortening release cycles from months into hours. Applications with high frequency releases often rely heavily on automated deployment tools using cloud infrastructure APIs. We report some results from experiments on reliability issues of cloud infrastructure and trade-offs between using heavily-baked and lightly-baked images. Our experiments were based on Amazon Web Service (AWS) *OpsWorks* APIs and configuration management tool *Chef*. As a result of our experiments, we then propose error handling practices that can be included in tailor-made continuous deployment facilities.**

Keywords: release engineering; system administration; continuous deployment; DevOps; continuous delivery

I. INTRODUCTION

The continuous delivery trend is reducing release cycles from months to days or even hours. For example, Etsy.com had 4004 releases into the production environment in 6 months, at an average 10 commits per release [1]. These high frequency releases often rely on cloud infrastructure APIs and virtual machine images for initial provision and then deployment tools to complete the deployment or upgrade. However, this high frequency introduces reliability challenges. In the past, infrequent deployment or upgrades were often done during scheduled downtime with careful monitoring of their execution. Minor reliability issues of the cloud infrastructure APIs did not pose a significant threat since there was sufficient time allocated to resolve minor issues. This is no longer true if these APIs are called tens of thousands of times per day for continuous delivery.

Virtual machine images can be provisioned in several forms. One is for the image to include *all* of the software ultimately to run in an instance. This is called "heavily-baking" the image (also called immutable/phoenix servers as these servers are expected not to be changed after booting). Another form for provisioning is to provision a *portion* of the necessary software, e.g. the operating system plus some middleware, and have the instance itself load the remainder of the necessary software once it has been instantiated. This is called "lightly-baking" the image.

A common industry solution to address reliability issues in continuous delivery is to use pre-baked images to replace existing VM instances. However, there is still significant debate around the extent of the baking. That is, should heavily-baked images be used or lightly-baked images [2]? The heavily-baked approach may build up a significant time overhead, since even minor changes warranting a release require preparing a complex image. The resulting large number of images have to be stored and managed creating an "image sprawl" problem. An application may also consist of many images where different images need to correspond to each other in some way. This often requires significant coordination thus delaying the deployment. On the other hand, the lightly-baked approach introduces more reliability issues during the loading of remaining software as external services and software repositories may have reliability issues themselves or versions may have changed from one instance load to the next,

In this paper, we compare the two different philosophies from the perspective of infrastructure reliability by creating a tailor-made deployment program using AWS OpsWorks APIs. We report the reliability issues and tradeoff between the two approaches. We propose some error handling practices and ways to validate outcomes of intermediary steps, so that errors can be detected early.

II. MOTIVATING EXAMPLE- ROLLING UPGRADE

We start with a motivating example, *rolling upgrade*, which is arguably the most important operation for continuous delivery and high frequency releases.

Assume an application is running in the cloud. It consists of a collection of virtual machine instances instantiated from a smaller number of different virtual machine images. A new machine image representing a new release for one of the images (VM_R) is available for deployment. The current version of VM_R is V_A and the goal is to replace the N instances currently executing V_A with N instances executing the new version V_B. A further goal is to do this replacement while still providing the same level of service to clients of VM_R. That is, at any point of time during the replacement process, at least N instances running some version of VM_R should be available for service.

One method for performing this upgrade is called *rolling upgrade*[3]. In a rolling upgrade, a small number of k instances at a time currently running version V_A are taken out of service and replaced with k instances running version V_B. The requirement for N instances running some version of VM_R can be met by creating k additional instances

running V_A as are simultaneously being upgraded. That is, by overprovisioning for the upgrade. The time taken by the replacement process is usually in the order of minutes. Consequently performing a rolling upgrade for 100s or 1000s of instances will take on the order of hours. The virtue of rolling upgrade is that it only requires a small number of additional instances to perform the upgrade. According to [3], rolling upgrade is the industry standard method for upgrading instances.

There are three categories of failures that can occur:

1. *Provisioning failure*. A provisioning failure occurs during the replacement process. Specifically, a provisioning failure occurs when one of the upgrade steps produces incorrect results. This paper examines these failures by comparing reliability rates between heavily and lightly baked images.
2. *Logical failure*. A logical failure is a failure due to the application being upgraded. Examples include version incompatibility or inter-instance dependencies. These failures are application-specific, and are not discussed in this paper.
3. *Instance failure*. Instance failure is a normal occurrence in the cloud. An instance failure can be caused by the failure of the underlying physical machine, a failure of the network, or a failure of a (networked) disk. This paper does not examine these failures, as they are not specific to deployment / upgrades.

III. OBSERVED CHALLENGES

We first implemented a simple rolling upgrade prototype using AWS OpsWorks API. OpsWorks is Amazon's automated DevOps tool which integrates with Chef configuration management to fully provision an application services in automated ways. In OpsWorks, an application service has a set of lifecycle events which are associated with custom-built Chef recipes. By calling operations from the OpsWorks API and other AWS APIs, our prototype can replace a configurable number of old application service instances with new application service instances.

The rolling upgrade prototype is then configured to support *two different types of rolling upgrades*. One is the *heavily baked* approach using a custom AMI (Amazon Machine Image) with built-in recipes which largely perform OpsWorks-related agent setups and user configurations. The other is the *lightly baked* approach using a basic AMI with custom Chef recipes which add more customized actions for installing additional software . From an abstract viewpoint, the two approaches are performing the same task: to upgrade an application (for our experiments we used the Tomcat server) in a rolling upgrade fashion. We chose to upgrade an application server since it is located in the middle of the stack, with complex dependencies. We did an explorative study in a cluster of 18 servers and then in a cluster of 72 servers for experiments. The granularity k of the rolling upgrade is set to be 3.

By comparing Figure 1 with Figure 2, we see that lightly baked image approach is less reliable than heavily baked image approach.

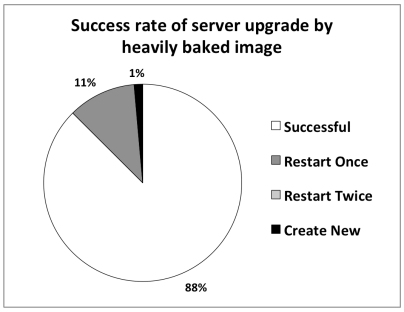

Figure 1. Success rate by heavily baked image approach

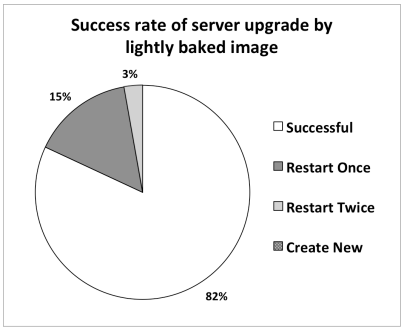

Figure 2. Success rate by lightly baked image approach

We recorded the time distribution for four different phases of the upgrade process. The four portions are stopping the current instances, pending, booting, and running:

stopping - Our prototype has called the AWS to stop an instance and is waiting for it to be stopped.

pending - AWS OpsWorks is waiting for the Amazon EC2 instance to start.

booting - The Amazon EC2 instance is booting.

running_setup - AWS OpsWorks is running the various Chef recipes.

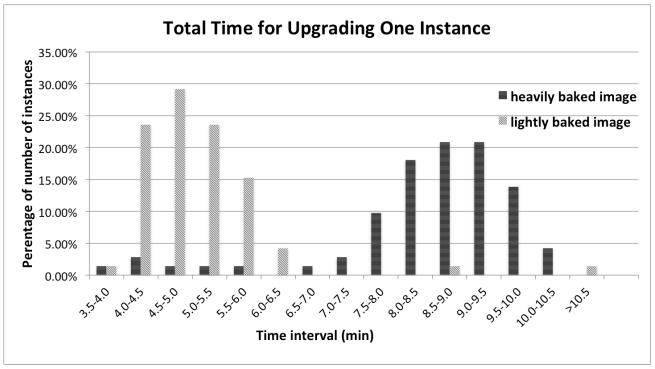

Figure 3. Total Time Distribution

Figure 3 shows the total time distribution. We found that most of the instances are upgraded in 4 to 6.5 minutes by lightly baked approach. However, it takes 8 to 10.5 minutes in upgrading most of the instances by heavily baked approach. It also shows that completion time distribution of the lightly baked approach has significantly longer tails than that of the heavily baked approach. This demonstrates that the heavily baked approach is more stable during upgrade although it takes a bit longer.

The next four figures show the breakdown time distribution for those phases.

For stopping and pending status, Figure 4 and 5 show that the time distributions for lightly and heavily baked approaches are relatively close on each time interval.

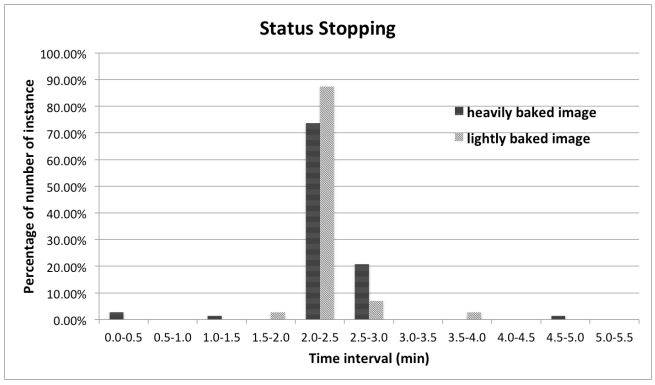

Figure 4. Stopping Status Time

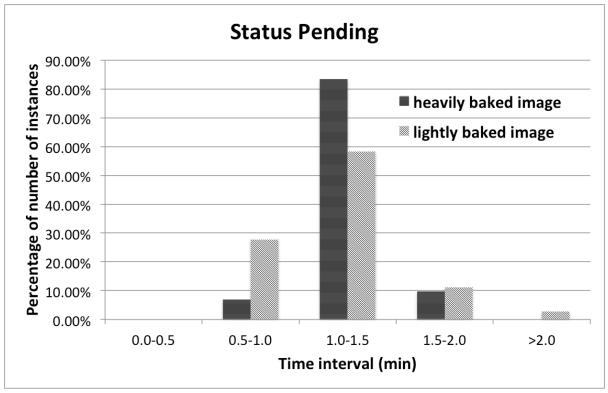

Figure 5. Pending Status Time

For booting status, Figure 6 surprisingly shows that the heavily baked approach takes significantly more time. The document on what exactly OpsWorks does to the instance during this phase is sparse other than it largely installs the OpsWorks agent and people reported some problems in this phase on the support forum.

As expected, Figure 7 shows longer time for lightly baked approach due to unreliable on-demand installation and configuration of the service.

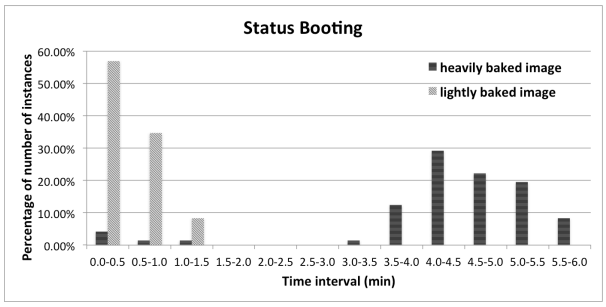

Figure 6. Booting Status Time

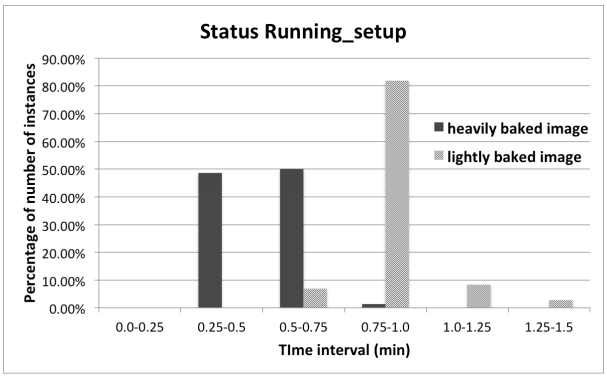

Figure 7. Running_Setup Status Time

In order to better understand the sources of unreliability, we compared the AWS-related API calls and Chef recipe size between the two approaches. For API calls, we enabled Amazon CloudTrail which logs all AWS API calls. The lightly-baked approach triggered 40 API calls while the heavily baked approach triggered 37. The difference is not material in terms of AWS basic infrastructure contributing to unreliability. For Chef recipe, the heavily baked approach has 69 actions per upgrade while the lightly baked approach has 142 actions. And many of the errors come from Chef and the OpsWorks agent which is a wrapper around Chef agent. We believe the additional Chef/OpsWorks related actions in the lightly-baked approach are the major contributors to slower time and unreliability.

IV. Discussion and Threats to Validity

First, some instances turn from pending to stopped (instead of the expected in-service state) in our explorative study. Our rolling upgrade prototype tried to restart these instances – however, we observed a high failure rate during restart. Hence we decided to rarely restart an instance but to start a new one to reduce the failure rate.

Second, OpsWorks did some major quality improvement over the 3-month period of our research. Some early mysterious errors such as “setup failed” disappeared and new mysterious errors around “update_custom_cookbooks” appeared. When we say mysterious errors, we mean we were unable to find the root cause of the error, but the execution succeeds in most deployment and fails some time

without obvious reasons. We posted the errors on the OpsWorks support forum and noticed many other reports of the same errors and experiences from other users.

Third, for the lightly-baked approach, we initially tried to directly trigger the undeploy event to remove old version of Tomcat. We observed a high failure rate. Also considering the best practices of reducing configuration drift and server life, we used stop-restart to trigger the undeploy and deploy for Tomcat whose upgrade frequency is less than a typical application. We did use deploy event to upgrade pure applications without restarting a server.

There is also a limitation in our comparison: it does not consider the image preparation time. The time may differ significantly between the heavily baked approach and lightly baked approach. In the past, the images were often prepared by starting a single instance, installing the required software and resealing them as images before provisioning them. The time and reliability issue is a major concern. However, the improved current practice is to use a *baking instance*[5] that modifies a mounted image directly to significantly speed up the preparation. The process is also more reliable as the image was never started with on-demand configuration to get its required software. We believe the baking time and reliability issue is largely resolved in practice.

There are some related works. When a release process is automated through scripts, the error handling mechanisms within the scripting or high-level languages can detect and react to errors and reliability issues through exception handling, for example error handlers in Asgard's Netflix [5] and Chef's fault handlers [6]. These exception handing mechanisms are best suited for a single language environment but continuous delivery often has to deal with different types of error responses from different systems. Exception handling also only has local information rather than global visibility when an exception is caught. Thus, external validation checks based on more global information are useful.

V. Proposed Solution

We now introduce some early solutions to the problems.

First, we deal with the *reliability issues* by incorporating fail fast, retry, and alternative actions in rolling upgrade tools itself. In our rolling upgrade prototype built on top of AWS OpsWorks, we implemented the following error detection and handling mechanisms.

- The prototype actively tracks the status of each instance through the life cycle and the time spent in each stage of the life cycle. The information is then used some of the approaches described below.
- Asynchronous upgrade: for rolling upgrade granularity of k (k>1), we do not wait until each wave is finished before starting the next wave. Whenever a single instance is upgraded, one more instance is being terminated and replaced. The granularity number only ensures that there are always k servers being upgraded.
- Timeouts specific to each status are used to fail fast. We collected historical data for upgrades and use the 95 percentile as the default (but configurable) timeout.
- We provide stop-restart, replace, deploy without restart and directly triggering of life-cycle events as alternative actions for many actions.

Second, as OpsWorks uses Chef significantly and it is a significant unreliability contributor mentioned early, we also implemented mini-test [4] based validation of intermediary outcomes for the Chef part. In the past, these tests are used during development time. We are using them during production runs as it helps detect errors early. These tests go beyond what the Chef error report or logs are reporting. They validate the expected final outcomes, not only the inputs to a chef execution or the execution itself.

VI. Conclusion

In this paper, we first investigated infrastructure reliability issues in high frequency releases on Amazon EC2. We compared upgrades based on heavily-baked vs. lightly-baked images by implementing a rolling upgrade prototype in AWS OpsWorks. We proposed some initial solutions on implementing one's own continuous deployment facility. We are also working on a framework, called Process-Oriented Dependability (POD), that works with existing deployment tools by analyzing logs produced by them [7]. It would be interesting to compare our results with results derived from other cloud providers but we have not done that, as yet.

Acknowledgment

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

References

[1] Etsy.com. (2013). *Managing Experimentation in a Continuously Deployed Environment*. Available: http://www.slideshare.net/InfoQ/managing-experimentation-in-a-continuously-deployed-environment

[2] P. Gillard-Moss. (2013). *Machine Images as Build Artefacts*. Available: http://peter.gillardmoss.me.uk/blog/2013/12/20/machine-images-as-build-artefacts/

[3] T. Dumitras and P. Narasimhan, "Why do upgrades fail and what can we do about it?: toward dependable, online upgrades in enterprise system," in *Proceedings of the ACM/IFIP/USENIX 10th international conference on Middleware*, Urbana, IL, USA, 2009, pp. 349-372.

[4] Calavera. (2014). *Minitest Chef Handler*. Available: https://github.com/calavera/minitest-chef-handler

[5] Netflix. (2013). *Aminator*. Available: https://github.com/Netflix/aminator

[6] OpsCode. (2014). *Chef*. Available: http://www.getchef.com/chef/

[7] X. Xu, L. Zhu, I. Weber, L. Bass, and D. Sun, "POD-Diagnosis: Error Diagnosis of Sporadic Operations on Cloud Applications," in *Dependable Systems and Networks (DSN), 44th Annual IEEE/IFIP International Conference on*, 2014, to appear.