Achieving Reliable High-Frequency Releases in Cloud Environments

Liming Zhu, Donna Xu, An Binh Tran, Xiwei Xu, Len Bass, Ingo Weber, and Srini Dwarakanathan, National ICT Australia

// Cloud applications with high-frequency releases often rely heavily on automated tools and cloud infrastructure APIs, both of which raise reliability issues. Experiments show that tradeoffs are also involved in the choice between heavily and lightly baked virtual-image approaches. //

cover image here

CONTINUOUS DELIVERY IS reducing release cycles from months to days or even hours. For example, Etsy.com had 4,004 releases into the production environment in six months, with an average of 20 releases per day and 10 commits per release.[1] Such high-frequency releases often rely on cloud infrastructure APIs and virtual machine (VM) images for initial provision, and then on deployment-related tools to complete the deployment or upgrade. However, this high frequency introduces reliability challenges.

In the past, developers often conducted infrequent, carefully monitored deployment or upgrades during scheduled downtime. Adopting cloud computing has given developers the opportunity to automate these tasks and moderately increase the release frequency. Previously, small reliability issues with the cloud infrastructure APIs or automated tools didn't pose a considerable threat because sufficient time existed to resolve the occasional issue. However, this is no longer true when—as our observations confirm—these APIs are called thousands of times a day and automated tools are used for true continuous delivery.

Applications in the cloud typically run on VMs, which are instances launched from a VM image, which typically takes one of two forms:

- *Heavily baked images*. The image includes all the software and most (if not all) of the configuration to run in an instance. (This form might also refer to heavily baking the immutable or phoenix servers, which aren't expected to change after booting, to prevent configuration drift.)
- *Lightly baked images*. The image contains only some of the necessary software, such as the OS and middleware. Each instance must load the remainder of the necessary software after being launched.

Considerable debate remains around how much baking is necessary, which further contributes to the reliability issues.

Here, we compare these two philosophies and report on specific reliability issues and tradeoffs. We identify major contributing factors at both the cloud-infrastructure and deployment-tool levels. Our general finding is that the more external resources you involve in a deployment—and the more you ask of those resources—the more likely you'll experience errors or delays. We propose various error-handling

 0740-7459/15/$31.00 © 2015 IEEE

practices and ways to resolve the issues, including cloud API wrappers and intermediary outcome validation to detect errors much earlier.

Motivating Example: Rolling Upgrades

Assume an application running in the cloud consists of a collection of VM instances, instantiated from a few different VM images. A new machine image representing a new release for one image (VM_R) is available for deployment. The current version of VM_R is V_A; 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 providing the same service level to VM_R clients. That is, at any point during replacement, at least N instances running some VM_R version should be available for service. One way to achieve this is with a *rolling upgrade*.[2]

A rolling upgrade takes out of service a small number of k instances at a time currently running V_A and replaces them with k instances running V_B. This technique can meet the requirement for N instances running some version of VM_R by creating the same number (k) of additional instances running V_B as are simultaneously being upgraded—thus overprovisioning for the upgrade. The replacement usually takes on the order of minutes. If k is set to 10 percent of N, a rolling upgrade can usually take less than an hour—even for hundreds or thousands of instances. Rolling upgrades are popular; their virtue is that they require only a few additional instances.

During an upgrade, three categories of failures can occur. *Provisioning failures* occur during replacement—specifically, when one upgrade step produces incorrect results. Here, we examine these failures by comparing reliability issues during provisioning using heavily and lightly baked images.

Logical failures are related to the application being upgraded, such as version incompatibility or interinstance dependencies. These failures are application-specific; we don't discuss them here.

Instance failures are a normal occurrence in the cloud. They can be due to failure of the underlying physical machine, the network, or a (networked) disk. Because they're not specific to deployment and upgrades, we don't discuss them here. They might occur within or outside the rolling-upgrade period and can be dealt with using traditional fault-tolerance mechanisms.

> The more external resources you involve in a deployment, the more likely you'll experience errors or delays.

Experiments and Observations

We performed rolling upgrades using Amazon Web Services (AWS) Elastic Compute Cloud (EC2) and OpsWorks.

OpsWorks is Amazon's automated DevOps (development and operations) tool, which integrates with Chef configuration management to fully provision an application service. In OpsWorks, an application service has a set of life-cycle events associated with custom-built Chef recipes. By calling operations from the OpsWorks API and other EC2 APIs, we can replace a configurable number of old application service instances with new ones. We're dealing with both cloud infrastructure APIs (EC2 APIs) and deployment software (OpsWorks, Chef tools, APIs, and so on). Both types contribute to the upgrade's reliability.

We configured the rolling upgrade to support both the heavily and lightly baked approaches. Heavily baked upgrades used a custom Amazon Machine Image (AMI) with built-in recipes, which largely performed OpsWorks-related agent setups and simple default configurations. Lightly baked upgrades used a basic AMI with more complex custom Chef recipes, which performed more customized actions for installing additional software. These upgrades required more actions after instantiation and relied much more on the deployment software (OpsWorks and Chef) than the heavily baked upgrades.

From an abstract viewpoint, the two approaches perform the same task: a rolling upgrade of an application. In our experiments, the application stack was a standard three-tier (Web, application, and database server) deployment of a content management system. We upgraded an application server (Tomcat) because it was located centrally in the stack with complex dependencies. We performed rolling upgrades to a cluster of 72 servers, with k set to 3, and recorded timings of different phases and reliability problems in each case.

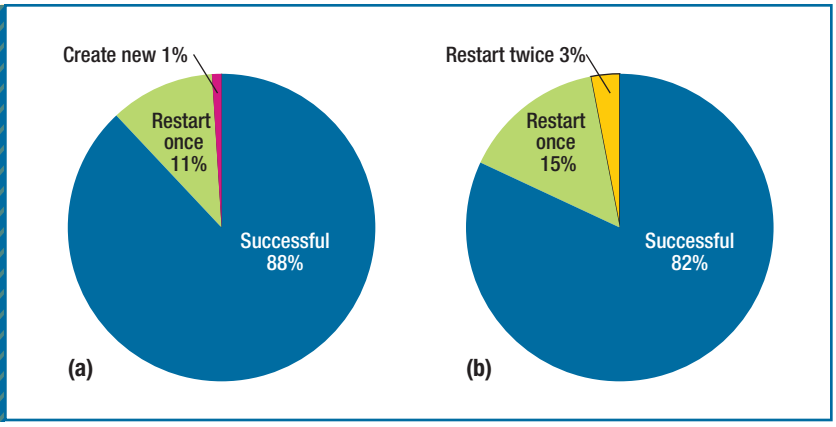


FIGURE 1. Success rates for (a) heavily and (b) lightly baked upgrades. Heavily baked upgrades were generally more reliable.

Figure 1 shows that, overall, lightly baked upgrades were less reliable than heavily baked upgrades.

We recorded the time distribution for the four upgrade phases:

- *Stopping or terminating*. The API operation for stopping or terminating an instance was called.
- *Pending*. OpsWorks is waiting for a new EC2 instance to start.
- *Booting*. An EC2 instance is booting.
- *Running setup*. OpsWorks is running Chef recipes.

Figure 2 shows the time distribution. We performed statistical tests and observed statistical significance in the timing profiles and all other reported timing differences. We have two key observations. First, lightly baked upgrades usually completed faster, typically in 4 to 6.5 min. Most heavily baked upgrades took 8 to 10.5 min.

Second, lightly baked upgrades had a broader distribution than heavily baked upgrades. The lightly baked upgrades' completion times could be significantly longer; their distribution had considerably longer tails (see the sidebar).

These two observations demonstrate that heavily baked upgrades were stable but usually took longer.

Figure 3 shows the time distributions for the four upgrade phases. For stopping and pending, the distributions for lightly and heavily baked upgrades are relatively close on each time interval. For booting, heavily baked upgrades take considerably longer. What exactly OpsWorks does

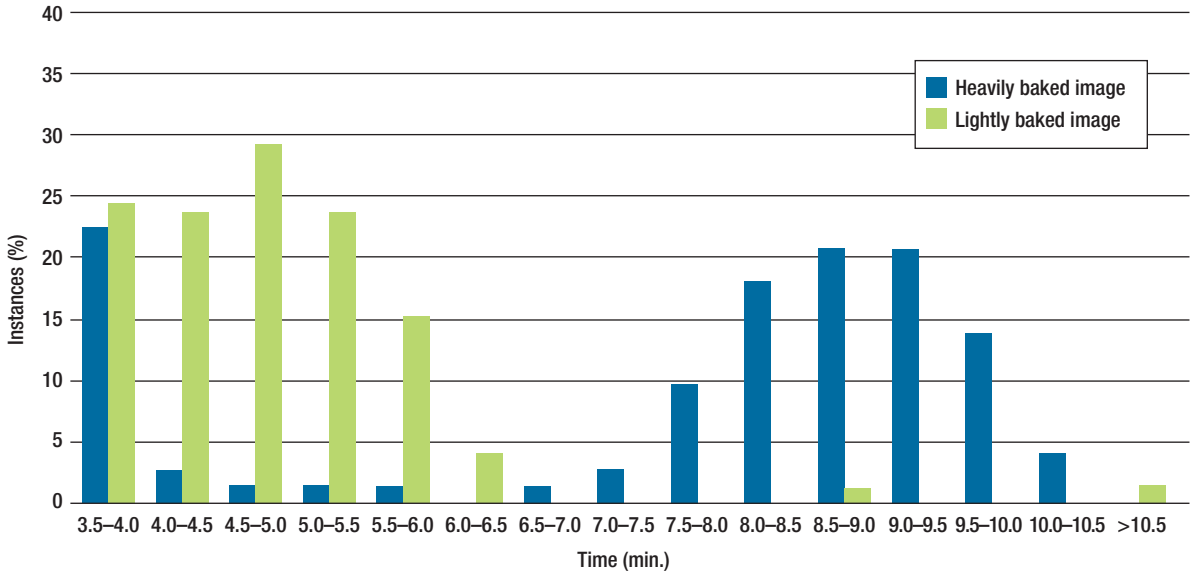


FIGURE 2. The time distribution for upgrading one instance. Lightly baked upgrades usually completed faster, with most instances taking 4 to 6.5 min.

with an instance in this phase isn't clear: the documentation states only that the OpsWorks agent is installed. Users reported various problems in this phase on the support forum. Lightly baked upgrades (including booting and running setup) can take longer, which might be attributable to unreliable on-demand service installation and configuration.

To better understand the unreliability's sources, we compared the AWS-related API calls and Chef recipe size between the two approaches. To capture the API calls, we used Amazon CloudTrail, which can log all AWS API calls. Lightly baked upgrades triggered 40 EC2 API calls,

THE LONG TAIL

A probability distribution has a long tail if a larger proportion of the population resides in the tail than there would be under a normal distribution. In other words, the 68–95–99.7 rule states that the expected percentages of population lie within one, two, or three standard deviations from the mean in a normal distribution, respectively. If the proportion is larger than these percentages, the tail is long. This often has considerable implications for large-scale applications;[1] such implications extend over the large-scale applications themselves onto operations on these applications.[2]

References

1. J. Dean and L.A. Barroso, "The Tail at Scale," *Comm. ACM*, vol. 56, no. 2, 2013, pp. 74–80.
2. X. Xu et al., "POD-Diagnosis: Error Diagnosis of Sporadic Operations on Cloud Applications," *Proc. 44th Ann. Int'l Conf. Dependable Systems and Networks*, 2014, pp. 1–12.

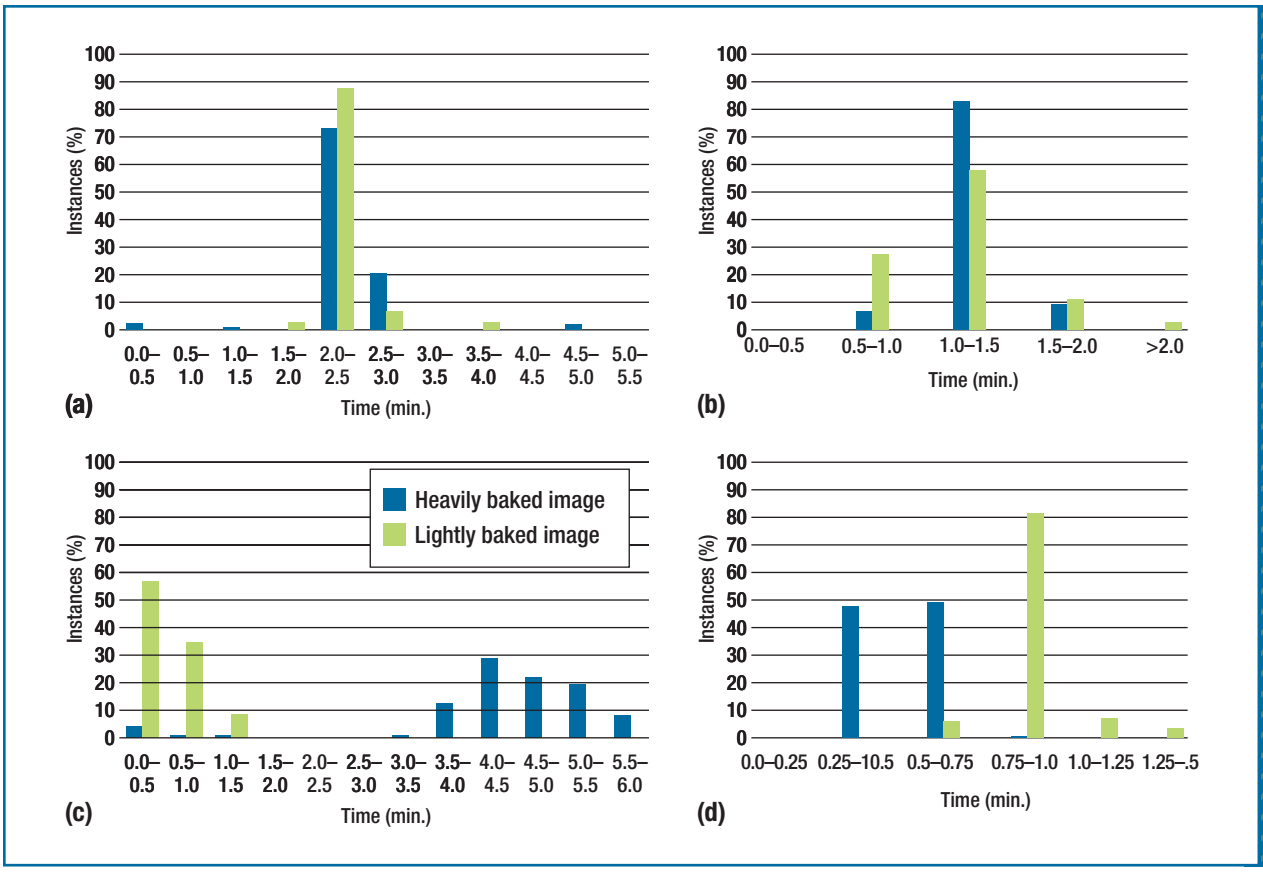


FIGURE 3. Time spent per status. (a) Stopping. (b) Pending. (c) Booting. (d) Running setup. For booting, the heavily baked upgrades took considerably longer; OpsWorks users also reported various problems during this phase in the user support forums.

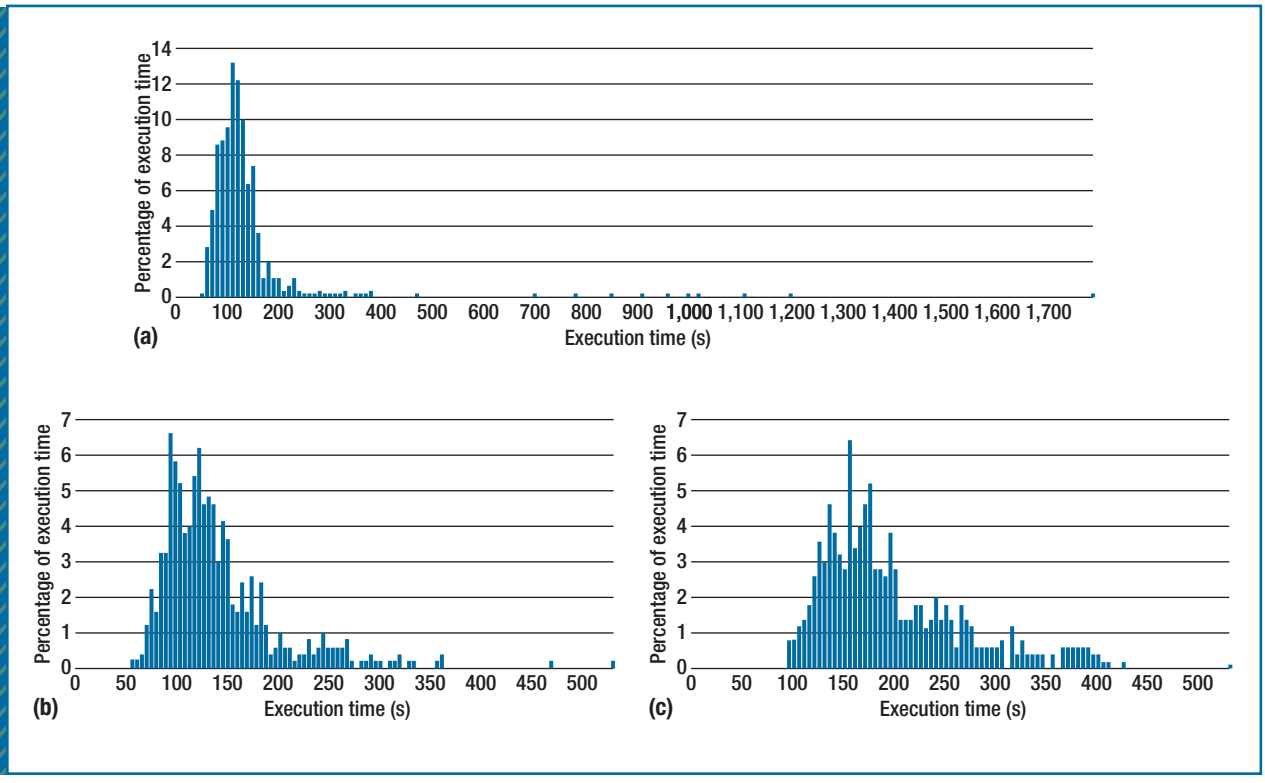


FIGURE 4. The execution time for the major steps of lightly baked upgrades. (a) **update_custom_cookbooks**. (b) **execute_recipes**—uninstall Tomcat 6. (c) **execute_recipes**—install Tomcat 7. All three steps clearly showed a long tail.

whereas heavily baked upgrades triggered 37. This difference isn't material in terms of the AWS basic infrastructure contributing to unreliability. For the Chef recipes, heavily baked upgrades had 69 OpsWorks API calls per upgrade, whereas lightly baked upgrades had 142.

Many of the errors came from Chef and the OpsWorks agent that acted as a wrapper around the Chef agent. We thus suspect that the additional Chef- or OpsWorks-related actions in the lightly baked upgrades were the major contributors to slowness, the long tail, and unreliability. Next, as we describe later, we investigated the factors and tried to establish whether many of the additional actions followed the long-tail characteristics in completion time.

During lightly baked upgrades, Chef's actions constituted three main steps:

1. Execute the **update_custom_cookbooks** deployment action. This step updated new custom Chef cookbooks from an external Git repository to the local instance cache over the network. In our case, the recipes were about the installation of a new Tomcat version (from version 6 to 7).
2. Execute the uninstallation-related **execute_recipes** deployment action. This ran the **tomcat::uninstall** Chef recipe to remove the old Tomcat version.
3. Execute the installation-related **execute_recipes** deployment action. This ran the **tomcat::setup** and **tomcat::configure** Chef recipes to install and configure the new Tomcat version.

As Figure 4 shows, all three steps clearly showed a long tail. On further examining the time stamps in logs, we observed two major contributors to the long tails.

The first surprise is the delay in action commands being acknowledged, executed, and reported, which contributed approximately 28 percent to the overall long-tail characteristics. Figure 5 shows the three delay types happening during each command's execution (excluding the execution itself):

- The delay from when the command was created until AWS

acknowledged it for execution (labeled A in Figure 5). In our observations, this delay was large (from 10 to 300 seconds).
- A small delay from when AWS reported to have acknowledged the command and Chef actually started running on the instance (we got this information from the Chef log). This delay is labeled B in Figure 5. It might have been caused by API calls or a delay in the OpsWorks agent triggering Chef on the instance. It was a few seconds at most.
- A small delay from when the Chef run completed on the instance until AWS reported that the command completed (labeled C in Figure 5). The delay time and reasons were similar to the previous small delay.

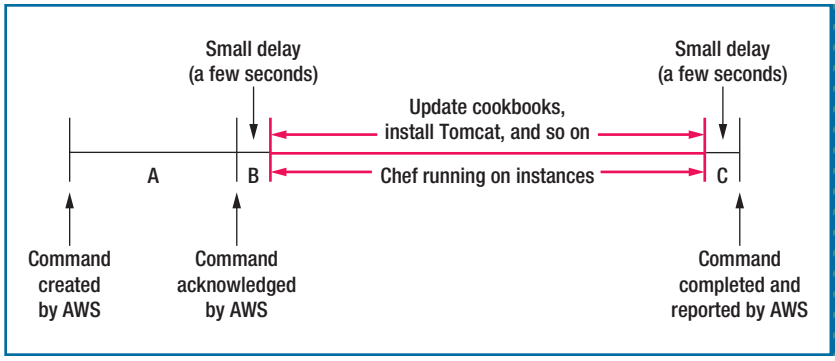


FIGURE 5. The three types of delays during each command's execution. Label A indicates the delay from when the command was created until AWS acknowledged it for execution. B indicates the delay from when AWS reported to have acknowledged the command and Chef actually started running on the instance. C indicates the delay from when the Chef run completed on the instance until AWS reported that the command completed.

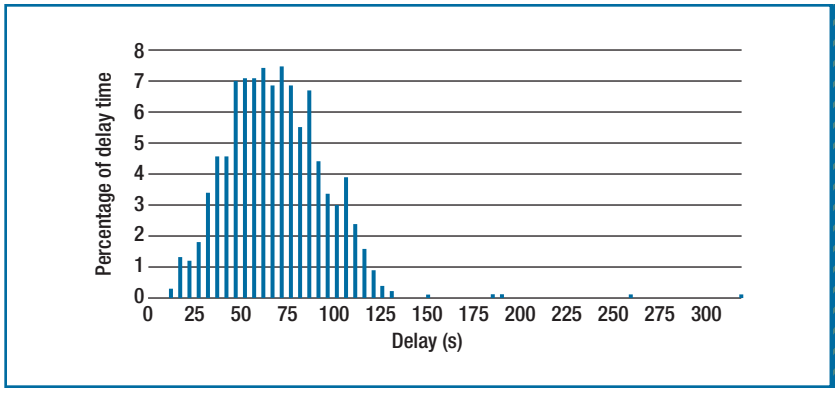


FIGURE 6. The total delay in command processing and reporting. This takes into account the three delays in Figure 5.

Figure 6 shows the time distribution for all three delays.

The second contributor to the delays stemmed from external resources—mostly, the software repository and dependency servers—which made up approximately 70 percent of the overall long-tail distribution. Downloading software from an external repository for installation or even resolving dependencies before removing an old version can have considerable long-tail characteristics. The remaining actions, such as configuring and updating cookbooks from local cache, show no such characteristics. Common industry practices use a local mirror server or some redundancy to relieve the issues around downloading from repositories.

One possibility is that the delays were due to Chef's somewhat unpredictable pull model, in which the configuration converges periodically toward the desired state. However, for OpsWorks and our rolling upgrade, this was transformed more into a push model: different types of Chef recipes were set to converge when triggered by specific conditions. For example, Chef recipes in OpsWorks' "configure" life-cycle event ran on all instances every time an instance in the OpsWorks stack entered or left the online state (for example, to start, stop, or terminate). So, Chef's pull model wasn't a major factor in this setup.

Discussion and Threats to Validity

This study has some obvious limitations. First, it built on AWS using OpsWorks and Chef, without testing other platforms.

Second, our comparison is limited: we didn't consider image preparation time, which might differ considerably between the heavily and lightly baked approaches. In the past, images were often prepared by starting an instance, installing the required software, and resealing it as

an image before provisioning. In that process, time and reliability were major concerns.

However, the improved current practice is to use a dedicated *baking instance*[3] that modifies a mounted image directly, considerably speeding up preparation. The process is also more reliable because the image is never started with on-demand configuration to get the required software. We believe the time and reliability issues are largely resolved in practice.

Third, 72 servers isn't a huge setup. We believe the outliers in the tail are proportional to the total number of servers upgraded in larger settings. But the outliers' effect on the total upgrade completion time might also depend on a rolling upgrade's granularity.

Only a few related approaches exist in this area. When a release process is automated through scripts, error-handling mechanisms in the scripting or high-level languages can detect and react to errors and reliability issues through exception handling. For example, error handlers in Netflix's Asgard (https://github.com/Netflix/asgard) and Chef (www.getchef.com/chef) react to detected errors. These exception-handing mechanisms are best suited for single-language environments, but continuous delivery often must deal with different types of error responses from different systems. Also, exception handling has only local information rather than global visibility when an exception is caught. So, external validation checks based on more global information are useful.

Our Proposed Solution

We now introduce some early solutions to the problems we've identified; our primary goal is to shorten the long tail in some operations.

To deal with the reliability issues, we incorporate fail fast, retry, and alternative actions in the rolling-upgrade tools. In our rolling upgrade using AWS OpsWorks, we implemented the following four error detection and handling mechanisms, which considerably reduced the reliability issues.

First, our system actively tracks each instance's status through the life cycle and the time spent in each life-cycle stage. The information is then used by approaches that we describe later.

> The reliability problems of deployment infrastructure and tools are considerable under the lightly baked approach.

Second, we implement asynchronous upgrades. For a rolling-upgrade granularity of k ($k > 1$), we don't wait until each wave is finished before starting the next wave. When a single instance has been upgraded and is online again, another instance is upgraded straightaway. The granularity then ensures that k servers are upgraded concurrently at any point in time. This also prevents recurring errors in an instance from blocking the whole upgrade. Should the problem recur on all subsequently started instances, the whole upgrade will fail-stop or be rolled back.

Third, we use time-outs specific to each status to fail fast. We collected historical data for upgrades and use the 95th percentile as the default (but configurable) time-out.

Finally, we provide stop–restart, replace, deploy without restart, and direct triggering of life-cycle events as alternatives for many actions.

In addition, OpsWorks heavily builds on Chef and, as we've seen, contributes considerably to unreliability. So, we implemented minitest-based validation (https://github.com/calavera/minitest-chef-handler) of intermediary outcomes for the Chef portion. Previously, these tests were used during development; we're using them during production runs to detect errors early. These tests go beyond what Chef error reports or logs are reporting. They validate the expected final outcomes—not just inputs to a Chef execution or the execution itself.

Finally, for EC2 API reliability issues, we analyzed their characteristics and implemented an API wrapper to solve the problem.[4]

As our investigation clearly shows, the reliability problems of deployment infrastructure and tools—excluding script and user errors—are considerable under the lightly baked approach and exhibit long-tail characteristics at scale. Key contributing factors here include infrastructure API reliability, command processing and reporting delays, and dealing with external resources. The mechanisms we described in this article are good strategies to alleviate these problems. Furthermore, we're developing Process-Oriented Dependability, a

framework that works with existing deployment tools by analyzing the logs they produce.[5]

The heavily baked approach can incur considerable preparation overhead because even minor changes warranting a release require preparing a complex image. The numerous images that result must be stored and managed, creating "image sprawl." Also, an application might comprise many different images that must correspond to each other in some way. This monolithic approach often requires considerable coordination, thus delaying deployment.

Acknowledgments

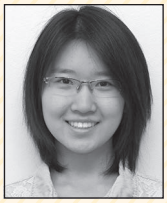
The Australian government's Department of Communications and the Australian Research Council's ICT Centre of Excellence Program fund National ICT Australia.

References

1. W. Stuckey, "Managing Experimentation in a Continuously Deployed Environment," 2013; www.slideshare.net/InfoQ/managing-experimentation-in-a-continuously-deployed-environment.
2. *Asgard*, Netflix, 2014; https://github.com/Netflix/asgard.
3. *Aminator*, Netflix, 2013; https://github.com/Netflix/aminator.
4. Q. Lu et al., "Mechanisms and Architectures for Tail-Tolerant System Operations in Cloud," *Proc. 6th Usenix Workshop Hot Topics in Cloud Computing*, 2014; www.usenix.org/conference/hotcloud14/workshop-program/presentation/lu.
5. X. Xu et al., "POD-Diagnosis: Error Diagnosis of Sporadic Operations on Cloud Applications," *Proc. 44th Ann. Int'l Conf. Dependable Systems and Networks*, 2014, pp. 1–12.

ABOUT THE AUTHORS

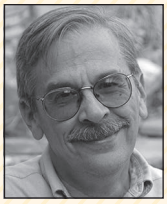
LIMING ZHU is a research group leader and principal researcher at National ICT Australia, and a conjoint lecturer at the University of New South Wales and University of Sydney. His research interests include software architecture and dependable systems. Zhu received a PhD in software engineering from the University of New South Wales. Contact him at liming.zhu@nicta.com.au.



DONNA XU is a vacation student at the Commonwealth Scientific and Industrial Research Organisation. Her research interests include cloud computing, dependability, and big-data processing. Xu received a BCST (Honors) in computer science from the University of Sydney. Contact her at donna.xu@nicta.com.au.

AN BINH TRAN is a research assistant at National ICT Australia. His research interests include development operations, cloud computing, dependability, and big data. Tran received a BSc (Honors) in computer science from the University of New South Wales. Contact him at anbinh.tran@nicta.com.au.

XIWEI XU is a researcher at National ICT Australia. Her research interests include dependability, cloud computing, development operations, and big data, as well as software architecture, business processes, and service computing. Xu received a PhD in software engineering from the University of New South Wales. Contact her at xiwei.xu@nicta.com.au.



LEN BASS is a senior principal researcher at National ICT Australia. His research interests include operating systems, database management systems, user interface software, software architecture, product line systems, and computer operations. He's coauthor of DevOps: A Software Architect's Perspective (Addison-Wesley, 2015), Software Architecture in Practice (Addison-Wesley, 2012), and Documenting Software Architectures: Views and Beyond (Addison-Wesley, 2010). Bass received a PhD in computer science from Purdue University. Contact him at len.bass@nicta.com.au.

INGO WEBER is a senior researcher in the Software Systems Research Group at National ICT Australia and an adjunct senior lecturer in computer science and engineering at the University of New South Wales. His research interests include cloud computing, development and operations, business process management, and AI. He's coauthor of DevOps: A Software Architect's Perspective (Addison-Wesley, 2015). Weber received a PhD in computer science from the University of Karlsruhe. Contact him at ingo.weber@nicta.com.au.

SRINI DWARAKANATHAN is a research intern at National ICT Australia. His research interests include distributed systems, cloud computing, high availability, and software-defined networking. Dwarakanathan received an MS in computer and information science from the University of Pennsylvania. Contact him at srini.nathan@nicta.com.au.