# Mechanisms and Architectures for Tail-Tolerant System Operations in Cloud

Qinghua Lu[1,2], Liming Zhu[2], Xiwei Xu[2], Len Bass[2], Shanshan Li[1], Weishan Zhang[1], Ning Wang[1]
[1]*College of Computer and Communication Engineering, China University of Petroleum*
[2]*Software Systems Research Group, NICTA*

## Abstract

Conducting system operations (such as upgrade, reconfiguration, deployment) for large-scale systems in cloud is error prone and complex. These operations rely heavily on unreliable cloud infrastructure APIs to complete. The inherent uncertainties and inevitable errors cause a long-tail in the completion time distribution of operations. In this paper, we propose mechanisms and deployment architecture tactics to tolerate the long-tail. We wrapped cloud provisioning API calls and implemented deployment tactics at the architecture level for system operations. Our initial evaluation shows that the mechanisms and deployment tactics can effectively reduce the long tail.

## 1. Introduction

Conducting system operations (such as upgrade, reconfig, deployment) for large-scale modern distributed systems in cloud is error prone and complex[1]. System operations in a cloud are performed through cloud APIs provided by cloud providers. Therefore, the completion time and reliability of these tasks depends on the reliability and performance of API calls. We previously did an empirical study on cloud API issues [4] and observed that a large percentage of the cases reported in the EC2 forum [5] are related to stuck API calls and slow response to API calls. The majority of the API issues are unavoidable timing failures that cannot be reduced in a large-scale system and often exhibit a crash-recovery behavior. Those API timing failures are major causes of the long-tail of the timing distribution of operation tasks. However, existing research on system operation focuses on reducing errors and repair time [2-3] rather than tolerating reduced latency issues.

From an architecture perspective, one step of an operation either needs to touch many cloud instances in parallel or results in "deep hierarchical" calls, which means one top level call leads to another call and another call. If one of these dependent calls is slow to respond in a large-scale system the initial operation will be slow to respond. Such problems are already being observed in typical large-scale fan-out systems. For example, Netflix Hystrix tries to keep timeouts short and to fail fast to avoid cascading timeouts [6]. Jeff Dean's request hedging technique makes the same request to multiple replicas and uses the results of the first request to respond [7]. We argue that large-scale deployment architecture in cloud is also a fan out and deep hierarchical system from an operational point of view and consequently deployment operations will also exhibit a long tail on their timing distribution.

In this paper, we propose a set of mechanisms and deployment architecture tactics to tolerate the long-tail. We implemented our mechanisms as a tail-tolerant wrapper around EC2 APIs which are heavily used in system operation of applications hosted in Amazon cloud. At the architecture level, we implemented the proposed deployment architecture tactics that also reduce the long tails. We evaluate our long-tail tolerant mechanisms and deployment tactics through a set of experiments on AWS infrastructure. Our initial result shows that the mechanisms and deployment architecture tactics can effectively remove the long tail of the timing distribution.

The rest of this paper is organized as follows. Section 2 presents the tail-tolerant mechanisms and our tail-tolerant API wrapper. Section 3 discusses the proposed deployment architecture tactics. Section 4 evaluates the proposed solutions. Section 5 covers related work. Section 6 concludes the paper and outlines the future work.

## 2. Tail-Tolerant Mechanisms and API Wrapper

An operation or a set of cloud API calls can be seen as a process or a workflow. Our approach for dealing with timing failures is to adapt exception-handling patterns of workflows [8-9]. We first discuss the workflow exception handling and then we discuss how we wrapped cloud API calls to utilize these patterns.

### 2.1. Tail-Tolerant Mechanisms

The workflow patterns that we are using assume there are six states within the lifecycle of a workflow operation: requested, cancelled, allocated, started, failed and completed. These are represented by the rectangles in Fig. 1. The transitions between the states are a combination of the workflow patterns and our adaptations. Our wrapper around the cloud API calls implements this state diagram.

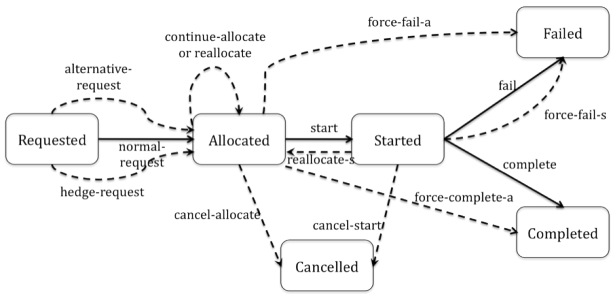


Figure 1. Failure/fault tolerance workflow patterns.

In general, the transitions represent calls to the original cloud APIs and the states represent either decision points or final states. The failed, completed, and cancel states represent final states and the other three represent decision points. The flow through this state diagram begins with an API call request that is intercepted by our wrapper which enters the Requested state. The Requested state may choose to make a normal request or a hedged request. For example, a hedged-request may issue one or more original EC2 API calls to launch instances. The wrapper then goes to the Allocated state. From the Allocated state, the wrapper may also force a complete or a failure depending on the state of operation. The solid arrows depict the state transition of an original EC2 API call during normal execution. The dashed arrows show the mechanisms of dealing a timing failure in a given state. Below we describe how we utilize these patterns in detail.

**When an API call is being requested**: The ***hedge-request*** pattern is similar to "hedges requests" idea in Jeff Dean's paper [7]. For certain operations (e.g. launching multiple VMs), we will issue more requests than we need (e.g. launching/scaling out 12 instead of the 10 we need) and then cancel the remaining immediately after the required number is successfully reached. In ***alternative-request*** pattern, an alternative API is requested at the same time at the same time as the original API is requested.

**When resources are being allocated to one or more original EC2 API calls**: ***Continue-allocate*** pattern, ***reallocate*** pattern, ***force-fail-a*** pattern, and ***force-complete-a*** pattern can be used when an API request sent to an instance (i.e. a Virtual Machine) fails or is unresponsive. The ***continue-allocate*** pattern schedules the request to be sent to the same instance at a future time if the API request fails or there is no response from the cloud infrastructure within a certain time. For example, in our disaster recovery product Yuruware Bolt, we need to move data from one region to another region for backup. One of the steps is to create an EBS volume from a snapshot. If the first "ec2-create-volume" call is failed or stuck, the application sends another "ec2-create-volume" request when a timeout occurs. The ***reallocate*** pattern, on the other hand, resends the request to other instances. For example, in Yuruware Bolt, we need two data mover instances in two regions for backup. The EBS volume created from the snapshot is required to be attached to an instance. If the EBS volume is not able to be attached to the instance, the application can attempt to attach the EBS volume to the instance again after several seconds using ***continue-allocate*** pattern or attach the EBS volume to the other available instances. The ***cancel-allocate*** pattern is used to cancel the volume allocation for the original instance.

Both ***force-fail-a*** pattern and ***force-complete-a*** pattern can be used when an API call has been retried for several times and continues to fail. A default fallback can be used by marking the call a failure or completion. ***Force-complete-a*** is useful when the output of the API call can be known from other operations. For example, after an instance is started, in the case that the command ec2-describe-instances does not return any output, the user could try to connect to the instance host. If the instance is accessible, it means the instance is running. Thus, the request to ec2-describe-instances can be force-completed and all subsequent API calls can be triggered. These patterns can be used to deal with unresponsive API calls and slow API responses.

**After an API call is started**: There are three patterns that can be used when an API call is stuck at a state, including ***reallocate-s*** pattern ***cancel-start*** pattern**,** and ***force-fail*** pattern. As we described earlier, an API call being stuck is a common complaint. In ***reallocate-s*** pattern, the application gives up the current request and restarts the API request again in another instance. For example, if an EBS volume cannot be attached to one instance, the application can try to attach it to a different instance within the same availability zone, and cancel the stuck attaching API call at the same time (***cancel-start*** pattern). Another example is that, the application can ignore the current request and resend the API request again to cloud infrastructure. For example, if an instance is stuck at initializing, the application can relaunch an instance. In ***force-fail*** pattern, if an API call is stuck at the state for a certain time, it is regarded as a failed API call and no subsequent calls are triggered. This pattern is similar to ***force-fail-r***.

### 2.2. API Wrapper

We implemented some of our mechanisms discussed in the previous section as a tail-tolerant API wrapper around Amazon EC2 APIs. The initial API wrapper only wraps around five API calls: launch an instance, start an instance, stop an instance, attach a volume and

detach a volume. These five EC2 API are the most frequently used and having significant latency issues according to our own experience and early empirical study of AWS developer forum [5]. We built a timing profile for each API call and resort to other means immediately after waiting time reaching a configurable 90 percentile of historical return time.

**Launch-instance:** The API wrapper implements the ***hedge-request*** pattern, which launches two instances through making two launch-instance API calls simultaneously when it receives a request. If one instance is launched within the time specified in the time profile of launch-instance, the API wrapper will kill the other one when it is launched. If neither of them launches, then the API wrapper implements ***continue-request-s*** pattern, which re-launches another two instances.

**Start-instance:** The API wrapper implements the ***alternative-request*** pattern, which starts an instance and launches a new instance using the same image simultaneously, and cancel the one with longer return time.

**Stop-instance:** The API wrapper launches a call to the stop-instance API, waits for the time specified in the time profile of stop instance. If the call is not completed, the API wrapper forces the instance to stop using the API of "force stop", which implements the ***force-complete-s*** pattern.

**Attach volume:** The API wrapper attaches volume to an instance and launches a new instance at the same time by using the ***alternative-request*** pattern. The wrapper waits for the time specified in the time profile of attach volume. If the call is not completed, it reattaches the volume to the newly launched instance.

**Detach volume:** The API wrapper waits for the time specified in the time profile of detach-volume. If not completed, then the API wrapper implements the ***force-complete*** pattern, which force-detaches the volume.

## 3. Deployment Architecture Tactics

Large-scale deployment architecture in cloud can be seen as a fan out and deep hierarchical system from an operation point of view. Deployment architectures tactics can remove the long-tail of operation tasks in cloud. Three industry best practices are adapted in our work to reduce the long-tail of operations in cloud.

**Immutable server:** During the provisioning of a service, an instance is first provisioned by launching a virtual machine image and then deployment tools are used to deploy software and configure the service in an on-demand fashion. A significant source of latency issues during a system operation comes from the on-demand phase after the server launches. Also, the longer a VM has been provisioned and running the more likely it is in an unknown state. Immutable server tactics means that operators make an image which contains everything a new version of an application needs. After the image is launched, nothing more is added or allowed to be changed. This can help reduce the tail latency issues during the on-demand phase.

**Micro services:** We break down an application stack or an application into smaller or even micro-services and make each service run on different VMs or lightweight containers. There are a number benefits in terms of reducing tail-latency. First, it significantly reduces the latency-causing-variability among the instances to be operated on. Instances belonging to a group to be operated on are essentially the same. Second, each instance is more lightweight. Third, there is less performance interference due to co-location. For example, traditionally, a single instance may have some 3 stateless services or a single service with the functionality of 3 web services. If operators want to upgrade something in one of the three web services, operators potentially introduce long tail because they have to touch a number of instances (say 100) and some of them will be slow statistically. With micro services, operators can have 100 instances for service 1, 100 instances for service 2 and 100 instances for service 3. For upgrade something, administrators touch 100 instances only and reduce the long tail probability.

**Redundancy:** Redundancy means administrator can run more than the required number of VMs to avoid long-tail operations. For example, if administrators want to upgrade 100 instances, to reduce long tail, during upgrade, administrators launch 103 instances as they expect at least 3 will be slow and unavoidable.

## 4. Evaluation

In this section, we evaluate our long-tail tolerant mechanisms and deployment architecture tactics through experiment.

### 4.1. Evaluation of Tail-Tolerant Mechanisms and API Wrapper

First we evaluate the API wrapper implementing the proposed API tail-tolerant mechanisms. Our experiments ran on AWS EC2. We selected the results of five API calls to report, including launch-instance, start-instance, stop–instance, attach-volume and detach-volume. For each API we wrapped, we measured the return time 1000 times respectively. Since the focus of this paper is long-tail, we removed the API calls that failed with error messages. The calculation of the percentage is still based on 1000 calls.

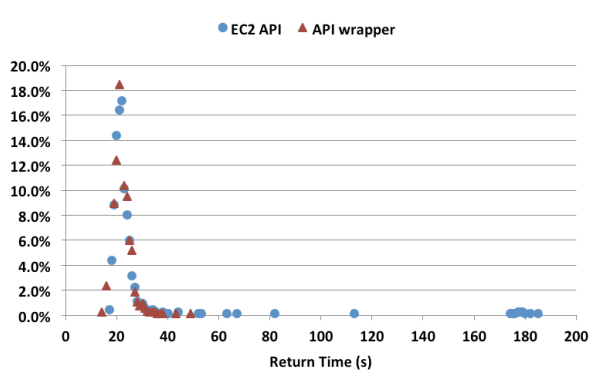


Figure 2. Measurement results of "start instance".

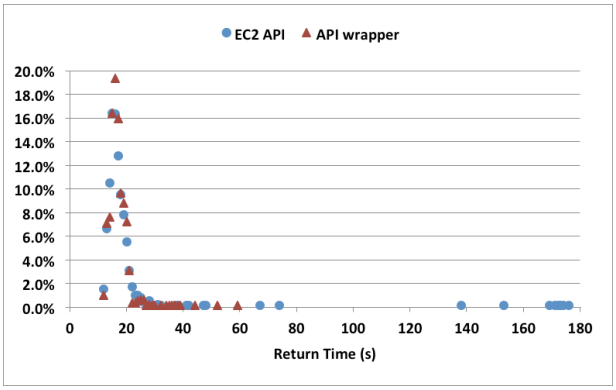


Figure 3. Measurement results of "stop instance".

We report the measurement result of start-instance and stop-instance in Fig. 2-Fig. 3. We omit the measurement result of the other three due to length limit. In Fig. 2-Fig. 3, the horizontal axis represents the return time of an API call while the vertical axis represents the percentage of the corresponding return time value among the return time of the total 1000 API calls. We observe that introducing tolerant mechanisms in our API wrapper significantly reduces the long-tail failure rate.

The longest return time of our API wrapper is 49s. The measurement result shows that the API wrapper and the original EC2 API have similar distribution of the return time when the return time is less than 49s. However, the original EC2 API has a long tail of the return time till 185s. API wrapper avoids 1.8% original EC2 API calls viewed as long tail (longer than 49s).

In Fig 3, 90.0% of original EC2 stop-instance API calls and 96.0% of stop-instance API wrapper return within 21s. 3.7% of stop-instance wrapper distributes from 22s to 59s. While there is 6.9% calls of the original EC2 API calls distributing from 22s to 59s, and the remaining 1.0% original EC2 API calls are long-tail till 176s as the longest return time.

### 4.2. Evaluation of Deployment Tactics

In this section, we evaluate the deployment tactics through automatically upgrading 50 AMP (Apache + MySQL + PHP) stacks by shell scripts. This experiment is running on AWS platform as well. We use Ubuntu Server 12.04.3 LTS as the operation system. The experiment upgrades the AMP stack from Apache 2.0.65, MySQL 5.1.73, and PHP 5.2.17 to Apache 2.2.22, MySQL 5.5.35, and PHP 5.3.10 respectively.

We implemented the three deployment tactics, and compared the number of the successful upgraded VMs using different deployment tactic with a baseline, which represents upgrade without any deployment tactics. Below are the detailed four cases in this experiment. 1) Baseline: we upgraded AMP running on 50 VMs to the recent versions directly on the original VMs. 2) Immutable server: we created an image of VM which runs the second version of AMP and launched 50 VMs using the image. Then we terminate the VMs running old versions of AMP. 3) Micro services: we ran Apache and PHP on 50 VMs and ran MySQL on another 50 VMs, then we upgraded them on the original VMs directly. 4) Redundancy: we launched 3 extra VMs with AMPs. After the 3 extra VMs are successfully launched, we started upgrading the 53 VMs with AMPs.

We ran each test cases 100 times. We compared the 4 test cases and observed the test results as shown in Fig. 4. The horizontal axis represents the number of VMs being successfully upgraded while the vertical axis represents the percentile of the corresponding VM number. Fig. 4 shows that all the deployment tactics could reduce the failure rate of upgrade. The reduction led by "micro services" and "redundancy" is not as much as the reduction led by "immutable servers".

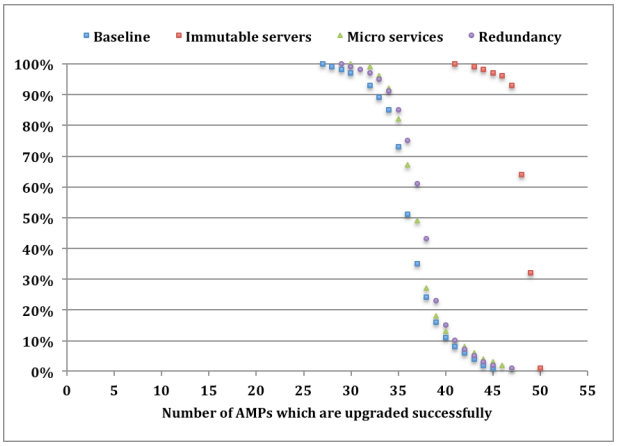


Figure 4. Measurement results of deployment tactics.

### 4.3. Discussion

Our experiments conducted in Section 4.1 show that our API wrapper with API tail-tolerant mechanisms can

largely reduce the long-tail of the original EC2 API. Although the probability of long-tail return time is very low, the time of long tail is very long. Sometimes it could be as long as 10 times of most of the return time. Our solutions can significantly reduce the impact of API issues on operations long-tail and improve the reliability of operations in cloud.

The experiment results discussed in Section 4.2 show that the proposed deployment tactics could reduce the failure rate of operation tasks in cloud both in parallel and in hierarchical. Through investigating the log produced during upgrade, we found that network problem causes most of the failures. Among the three deployment tactics, "immutable servers" has the largest impact on reducing the failure rate because launching instance prevents the VM failure due to the problem of network connection. "Micro services" reduce the failure rate because each service runs independently and performance inference can be avoided to a certain degree.

Users need to be aware: 1) some mechanisms/tactics will incur cost (e.g. micro resources in splitting AMP) which is clear from the configuration about percentage over provisioning or estimated tail-latency size; 2) effectiveness of our solution does depend on how the current operation is designed in terms of parallelism and VM dependency but we are agonistic to it by providing an API-level wrapper and optimization; 3) our solution is at the API wrapper level without requiring users to change their code calling the API. For supporting multiple cloud, we will have separate wrappers for different cloud providers. Our solution is not about a standardized API across cloud, which requires users to change their code. However, it is possible our mechanisms can be across cloud behind the scene.

## 5. Related Work

In cloud systems, runtime operation failures are due to different reasons [14-15], e.g. availability zone outages, hardware errors, overloaded database, operating system crashes [10], software bugs. Cloud infrastructure providers may not fully disclose the causes of outages or cloud infrastructure design for competitive reasons, which makes the study of API issues more important.

Microsoft researchers analysed cloud hardware failure and faults [11]: hard disks are the most frequent failed hardware due to its frequent usage and unreliability; 8% of servers in the data center can experience at least one hardware incident a year; if a failure happens, the occurrence rate of another failure in the same server is high. Gill et. al. [12] found that networks in data centers are highly reliable. However, load balancers experience many software faults and network redundancy is not entirely effective. Many of these failures are reflected differently at the API level where the users may not know the underlying causes.

A significant portion of the API issues is related to slow API responses. Dean from Google summarized the reasons of slow response API responses [7]: 1) different applications may reside upon one machine and share resources; 2) applications running on different machines may share global resources; 3) background programs may generate latency; 4) various queuing in network switches and intermediate servers may cause latency. Dean believes that resource over-provisioning, real-time engineering of software, and improved reliability can help reduce the causes of API call latency. However, it is impossible to eliminate all API call latency. Therefore, Google proposes two techniques to deal with API call latency [7]: 1) Within-request immediate-response technique that is to issue the request to multiple replicas and use the first responded results; 2) cross-request long-term adaptation technique that is to issue different requests to different partitioned data.

At the application deployment level, approaches like [13] were proposed to optimize reliability, latency and energy when application components are deployed onto physical machines. However, the deployment platform involves physical machines where one has full control/visibility rather than infrastructures with specific auto-scaling facilities and failures ranging from individual nodes to entire region.

## 6. Conclusions

In this paper, we proposed tail-tolerant mechanisms and deployment architecture tactics to tolerate long-tail issues of operations in cloud. We implemented our mechanisms as a tail-tolerant wrapper around Amazon cloud APIs which are heavily used in system operations of applications hosted in Amazon cloud. Our initial evaluation shows that the mechanisms and deployment architecture tactics can remove the long tail.

## 7. Acknowledgements

This project is supported by "the Fundamental Research Funds for the Central Universities" (No. 14CX02140A and No.14CX02137A) and "the Scientific Research Foundation of China University of Petroleum" (No. Y1307021).

NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

## 8. References


[1] R. Colville and G. Spafford, Configuration Management for Virtual and Cloud Infrastructures, http://www.rbiassets.com/getfile.ashx/42112626510

[2] Yuan, D. et al., "Be Conservative: Enhancing Failure Diagnosis with Proactive Logging," 9th ACM/USENIX Symposium on Operating Systems Design and Implementation (OSDI'12), Hollywood, CA, 2012.

[3] Chiarini, M. "Provenance for System Troubleshooting," 25th Large Installation System Administration Conference (LISA2011), Boston, MA, 2011.

[4] Lu, Q., Zhu, L. Bass, L., Xu, X., Li, Z., and Wada, H. 2013. Cloud API Issues: an Empirical Study and Impact. In Proceedings of 9th ACM SIGSOFT Conference on Quality of Software Architecture, pp.23-32.

[5] Amazon. 2013. Amazon Elastic Compute Cloud Forum. https://forums.aws.amazon.com/forum.jspa?forumID=30

[6] Netflix. 2013. Hystrix. https://github.com/Netflix/Hystrix

[7] Dean, J. and Barroso, L. A. The Tail at Scale. Communications of the ACM. vol. 56. pp. 74-80.

[8] Russell, N., Aalst, W. V. D. and Hofstede, A. T. 2006. Workflow Exception Patterns. In Advanced Information Systems Engineering. pp. 288-302.

[9] Russell, N., Aalst, W. V. D., Hofstede, A. T., and Edmond, D. 2005. Workflow Resource Patterns: Identification, Representation and Tool Support. In Advanced Information Systems Engineering. pp. 11-42.

[10] Ford, D., Labelle, F., Popovici, F. I., Stokely, M., Truong, V. A., Barroso, L. 2010. Availability in Globally Distributed Storage Systems. In Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation.

[11] Vishwanath, K. V. and Nagappan, N. 2010. Characterizing Cloud Computing Hardware Reliability. In Proceedings of the 1st ACM symposium on Cloud computing. pp. 193-204.

[12] Gill, P. 2011. Understanding Network Failures in Data Centers: Measurement, Analysis and Implications. In Proceedings of the ACM SIGCOMM 2011 conference. pp. 350-361.

[13] Malek, S., Medvidovic, N., and Mikic-Rakic, M. 2012. An Extensible Framework for Improving a Distributed Software System's Deployment Architecture. IEEE Transactions on Software Engineering. vol. 38. pp. 73-100.

[14] Lu, Q., Xu, X. Zhu, L., Bass, L., Li, Z., Sakr, S., Bannerman, P.L., Liu, A. 2013. Incorporating Uncertainty into in-Cloud Application Deployment Decisions for Availability. In Proceedings of 2013 IEEE Sixth International Conference on Cloud Computing (CLOUD'13), pp.454-461.

[15] Xu, X., Lu, Q., Zhu, L., Li, Z., Sakr, S., Wada, H., Webber, I. 2013. Availability Analysis for deployment of in-Cloud Applications. In Proceedings of the 4th International ACM Sigsoft Symposium on Architecting Critical Systems (ISARCS'13), pp.11-16.