FEATURE ARTICLE: CLOUD APIS

An Empirical Study of Cloud API Issues

Zhongwei Li and Qinghua Lu
China University of Petroleum

Liming Zhu and Xiwei Xu
Commonwealth Scientific and Industrial Research Organisation

Yue Liu and Weishan Zhang
China University of Petroleum

With the emergence of the DevOps movement, software engineers are starting to rely on cloud platform APIs for implementing many fault tolerance, self-adaptation, and continuous delivery features such as deployment changes, scaling out/in, exception handling, backup/recovery, and migration. In this article, we present an empirical study of API issues in commercial cloud platforms. We classify the API failures and their causes, and discuss possible remedies for each category to improve the reliability of cloud applications and introduce our solutions to deal with resource characteristics faults and late timing failures.

The DevOps movement in industry is giving software engineers more responsibilities in terms of using operation environment/platform (for example, Cloud) facilities to automate the deployment, configuration, and run-time environment error detection to enable better self-adaptation and fault tolerance of the application. Such deployment of an application into the cloud requires special programs that handle configurations and provisioning that are specific to the cloud platforms. Once the application is running and performing its normal activities, these programs, sometimes in collaboration with the application, heavily rely on cloud infrastructure APIs to perform sporadic activities such as deployment change, scaling out/in, fault tolerance, backup/recovery, and migration. Some operations are performed through a management console or scripts, but they all use cloud platform APIs behind the scenes to complete the operations. As cloud consumers have limited visibility and control of the public cloud infrastructure, cloud platform API calls are sometimes the only interaction points between the cloud infrastructure and the cloud applications.

Various online discussion forums and our own cloud product development experiences (www.yuruware.com) highlight many issues with these APIs especially when they are needed for reacting to rapid changes in the environment (for instance, outages or workload spikes) or dealing with error-prone operational processes (for instance, upgrade, backup, and recovery).

Copublished by the IEEE CS and IEEE ComSoc
2325-6095/18/$33.00 USD ©2018 IEEE

Many cloud applications are architected to achieve high availability through on-demand resource allocation, micro rebooting, replication, and failover across geographically distributed sites (techblog.netflix.com, www.yuruware.com). Most of these operations are also performed through API calls and often under stress conditions. The application availability depends on the availability of cloud resources and the reliability and performance of these API calls. Software engineers need to understand the nature of the potential faults with API calls and implement proper failure handling and fault tolerance mechanisms for dealing both with API faults and the underlying resources faults.

This work started as a small empirical study[1] of 5 Amazon Elastic Compute Cloud (EC2; aws.amazon.com/ec2) calls out of our own frustration in using them to develop a cloud management software. Cloud management software relies heavily on cloud APIs to perform operations such as snapshotting/starting/stopping VMs, attaching/detaching volumes to a VM to do analysis, and redeploying a system with properly adjusted security, network, and topology settings. The motivation of the study also came from the fact that a large percentage (53%) of the cases reported in the Amazon EC2 forum are related to API failures. In the study of Qinghua Lu and colleagues,[1] we searched only the EC2 forums for 5 highly used calls. We identified 922 cases and classified those using traditional failure and fault categories. One of our conclusions was that the traditional classification did not provide many insights into the cloud context and possible remedies.

In this article, we report on a broader empirical study of Cloud infrastructure API issues using publically available information. The sources include 32 cloud platforms' discussion forums (32 cloud platforms that provide similar APIs to Amazon EC2 to supplement the main findings), technical analysis of API issues during outages from reputable sources, such as Amazon outage reports (aws.amazon.com/message/65648/, aws.amazon.com/message/67457/, aws.amazon.com/message/65649/, aws.amazon.com/message/2329B7/, aws.amazon.com/message/680342/), Netflix technical blogs (techblog.netflix.com), and AvailabilityDigest (www.availabilitydigest.com) and our own experience on product development on Cloud (www.yuruware.com).

The major contributions of this article include the following:

- We extracted 2087 API failures from a wide range of sources (the 922 cases in our previous work are included).
- We classified the API failures using the traditional failure classification[2] in dependable computing. In addition, we discussed the error messages and different potential remedies for each category.
- We classified the faults (causes of these failures) using a new scheme. We have some initial recommendations targeted for each category.
- We discussed research needs, tooling needs, and some potential design remedies that can be used by cloud application designers.

METHODOLOGY

Study Sources

The study sources include API discussion forums and technical analysis of API issues during outages from reputable sources (such as Amazon outage reports, Netflix Tech Blog, AvailabilityDigest), which are all publically available information.

Our study considers 32 platforms (the first four are major sources): Amazon EC2, Abiquo, Bluebox, Brightbox, CloudSigma, CloudStack, Dreamhost, Enomaly, ElasticHosts, Eucalyptus, Gandi.net, GoGrid, Google Compute Engine, Joyent, IBM SCE, Linode, Nimbus, Ninefold, OpenNebula, OpenStack, OpenSource Cloud, Rackspace, RimuHosting, Slicehost, SoftLayer, Terremark, VMware vCloud, VCL cloud, Voxel, VPS.net, skalicloud, serverlove.

In addition to the above publicly available information, during the development and operations of our commercial cloud management product (www.yuruware.com), we learned many lessons with respect to EC2 API reliability and performance.

Study Procedures

Data Collection

As cloud storage issues are often very different from the cloud compute issues, the scope of this study is on cloud compute related operations such as describe, reboot, create, destroy, make snapshot/images virtual machines or nodes, and attach/detach volumes to an instance. We also cover some network related issues including IP management and load balancer management such as attach/release IP addresses from an instance, and create, describe, attach, and detach load balancers to a subnet and their configuration as these are often critical during outages.

Our study scope started January 2011; the evolution of the platforms may make some of the old cases irrelevant. We collected 2,087 failures and 1,466 faults in total. The majority of them come from Amazon EC2 (1,846 failures and 1,326 faults).

Failure and Fault Classification

As shown in Figure 1, we largely classified the API failures and faults using the taxonomy of dependable and secure computing[2] with important improvements making them more cloud specific. The percentage distribution of failure and fault categories gives some insights into the severity and prevalence of the issues while the categories themselves give inspiration on category-specific solutions especially from a cloud consumer perspective.

A failure is an event that occurs when the delivered service deviates from the correct service. We consider all failed or slow API calls as failures. We documented the symptoms of the failures and, if available, the error messages.

A fault is the adjudged or hypothesized cause of an error.[2] In this study, we try to locate the causes of a manifested failure. Not all failure cases have a cause discussion in the original thread. However, many failures are very similar to each other and happen in similar contexts. For example, a large number of API call failures were reported immediately after an outage. The final outage analysis reports then go into details discussing the causes of these API call failures. Some failure discussions point to other threads that may have a fault discussion. As we have analyzed a very large sample, we are in a unique position to link similar failure issues and reuse faults discussions in some of these linked failures to represent all of them. The number of fault cases is smaller than the number of failure cases as we could not locate some of failure causes.

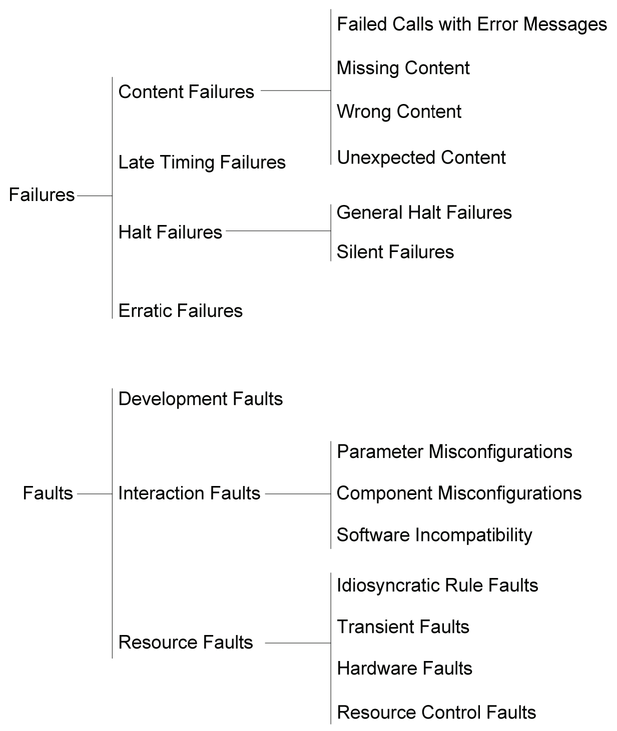

Figure 1. Classifications of failures and faults.

CLASSIFICATION OF API FAILURES

We classify the reported API failures into four major types: content failures, late timing failures, halt failures, and erratic failures. Among reported API failures, 55% are content failures, 35% are halt failures, 6% are late timing failures, and 4% are erratic failures.

Content Failures

Content failure is defined[2] as "the content of the information delivered at the service interface deviates from implementing the system function." We further classify content failures into four sub-types:

- failed calls with error messages (49%),
- missing content (2%),
- wrong content (2%), and
- unexpected content (2%).

Failed Calls with Error Messages

Only 49% of the overall API failures are failed calls with error messages. In these cases, 58% of the time, users understood the problem from the error message but did not know the right solutions to the problem, while 42% of the time, users could not pinpoint the problems from the error message.

Symptom: When a user tried to start an instance, the operation failed with an unclear error message.

Root cause: Unknown.

Solution: AWS engineers advised detaching the EBS volume from the instance and attaching it to another running instance.

Error message: State Transition Reason - Server.InternalError: Internal error on launch

In the above case, the user could not pinpoint the issue from an unclear error message. The failure could be caused by using the wrong name for the root volume. The root volume must be named /dev/sda1. Many people used /dev/sda instead, and it may generate the Server.InternalError message upon launch. Intuitively, the infrastructure software could know the problem precisely and generate a more specific error message. From a cloud consumer perspective, diagnosis or configuration checking tools using a knowledge base could have encoded the specific rule and helped diagnosis.

Symptom: Failed API calls with Request limit exceeded error message.

Root cause: API calls exceeded limit.

Solution: N/A. There is no official information on the limit or the time span on which the limit is calculated or suggested wait time.

Error message: Client.RequestLimitExceeded: Request limit exceeded

The returned error message in the case above pinpoints the problem. However, the users could not take specific actions. This is well known and studied in the bug community when an error message is helpful only to internal developers rather than developers who are interacting with the software. This is also true in misconfiguration studies[3] where people cannot act on a message. Or in the worst-case scenario, a specific error message misleads the user/developer to the wrong place.

Missing Content

This type of failure captures the cases where some information is missing in the returned output after an API call. Two percent of the cases are in this category, and most of them are related to "describe" type of calls, for example, ec2-describe-instances, which are used for monitoring.

Symptom: A user tried "ec2-describe-instances --filter "platform="" using the EC2 API tools and received nothing when he has some running Linux instances. In the API document, it says "Use windows if you have Windows based instances; otherwise, leave blank."

Root cause: API bug.

Solution: The AWS engineer forwarded this to the EC2 development team to investigate.

The case above shows an API failure case where the content is missing in the returned list of instances. A bug caused this particular case. The EC2 development team later found there was no easy way to filter Linux instances. The users either do not use the "platform" filter and it will return all instances or only use the windows filter. Despite the difficulties in reliably identifying Linux instances and the bug, returning the total number of running instances alongside of the empty filtered list could have helped the users diagnose the failure faster. Another popular case is related to transient missing information perhaps due to internal information propagation and eventual consistency. Cloud platforms should inform consumers that certain information might not show up immediately and give an estimated time for eventual consistency.

Wrong Content

Two percent of cases report that the output is wrong or the output is inconsistent from different calls. This usually happens when a user tries to detach a volume or describe an instance or a volume.

Symptom: The output of ValidFrom and ValidUntil are swapped in ec2-describe-spot-instances in ec2-api-tools 1.6.1.4.

Root cause: API bug.

Solution: Use defensive programming to correct the errors.

In the above case, an AWS engineer recommended using the latest version of the EC2 API tool "ec2-api-tools 1.6.1.4." The user came back and reported that the latest version had the same issue, but AWS did not follow up with the case.

Symptom: A user found that the output of ec2-describe-instances and ec2-describe-volumes did not match: the output of ec2-describe-instances showed the volume was attached while the output of ec2-describe-volumes showed the volume was attaching.

Root cause: N/A

Solution: Wait.

The case above shows that the outputs of two different API calls about the same information are inconsistent, which happens very frequently. In some cases, it is a transient problem and the answers will converge. In other cases, the inconsistency remains. This depends on the internal implementation of the APIs and how the state information is stored. For key operations, the cloud consumers cannot trust a single source of information, and in this case, the users have to SSH into the instance to find out the true state.

Unexpected Content

Two percent of the cases report the outputs of the calls are different from the users' expectation. Most of them are related to API calls describing the status of resources.

Symptom: A user launched an instance in the us-west-1 region. When the user ran the ec2-describe-instances command, it shows that the instance did not exist. When the user listed all instances, the output showed all of instances in us-east-1 region.

Root cause: Misconfiguration.

Solution: Using the --region option to specify us-west-1.

The common failure shown above is caused by user misconfiguration. Many users thought the ec2-describe-instance command returns instances in all EC2 regions, and they spent a significant amount of time diagnosing expectation mismatch. This is not technically a failure. However, the returned results could be more informative by stating that the current results contain only instances from a specific region. To have the original intentions of the call be part of the self-describing information of the returned results (rather than implicitly assumed) can help users diagnose issues.

Late Timing Failures

Late timing failure means that the arrival time of the delivered information at the interface deviates from implementing the system function,[2] but it does eventually arrive. Six percent of cases featured complaints about slow responses to users' API calls. As explained by the AWS support team, the slowest API calls can take several minutes to complete.

Symptom: It took 16 minutes for an instance to stop.

Root cause: N/A.

Solution: The AWS engineer advised to try "force stop" twice if this happens next time.

The above case shows a timing failure. Many users experience longer time to complete, for example, half hour or even a couple of hours. However, in the API document, it does not provide information on the normal time to complete each API call. This is understandable for some calls such as starting/stopping/snapshotting an instance, where the time highly depends on the characteristics of the instance. However, some information about typical instances could help. For other API calls such as attaching/detaching volumes or regarding load balancers or IP management, a general expectation on timing and recommended options for resolving late timing failures could be very helpful.

In large-scale distributed systems, latency variability is something that cannot be avoided but only tolerated. For example, Netflix Hystrix builds a timing profile on services and calls so that it can choose to fail fast after the waiting time has passed a certain percentile (for instance, 95%). We also built EC2 API call timing profiles during the development of our commercial product. We measured the 5 most frequent EC2 API calls ("launch instance," "start instance," "stop instance," "attach volume," and "detach volume") by calling the API 1,000 times and recording the return time under various realistic conditions.[5]

Halt Failures

Halt failures refer to cases where the external state becomes constant.[2] API calls in halt failures do not return while API calls in late timing failures do return after some time. A special case of halt failures is silent failures, which means no delivered service at all at the system interface. Thirty-five percent of the failures were in this category.

General Halt Failures

Table 1. General Halt Failures.

Intended call	Halted state	Percentage
Stop an instance	Stopping	63%
Start an instance	Initializing or pending	6%
Detach a volume	Detaching	31%

The most frequent API issue is that an API call is stuck at a certain state. Table 1 shows the percentage distribution of API calls stuck at different states.

Symptom: A user reported that the instance is stuck at stopping and "force stop" would not help.

Root cause: N/A.

Solution: The AWS engineer stopped the instance for the user on the AWS side.

Side effect: AWS killed the volume that the user was hoping to reuse.

Many users experience calls with stuck states subsequently. One example is shown above. It often happens that the AWS support team helps the user stop an instance at the AWS side, and the user comes back to complain that the instance is hanging at "initializing" or "pending" state when the user tries to start the instance.

On Amazon EC2, stopping/re-starting an instance changes the underlying physical machine for the instance. Users often try to stop and re-start instances after they cannot connect to their instances. After stopping the unresponsive instance and re-starting the instance, the instance can move from the unhealthy host to a healthy machine. If users are unable to stop their instances, they often try to detach their volumes and attach them to other instances.

Messages around stuck calls are generally non-existent and users can do very little. Experienced users have good snapshots and usually do not wait in such cases, as the possibility of recovery even with AWS assistance is very small.

Silent Failures

The second most frequent API failure category is unresponsive calls, which accounts for 18% of the cases.

Symptom: An instance was not accessible and the user could not stop/start it or creates a snapshot of it.

Root cause: AWS outage.

Solution: The AWS engineer advised that the user must launch a replacement instance from a pre-existing backup (EBS AMI). Attempts to stop an inaccessible instance will likely result in an instance becoming stuck in the stopping state. Customers that do not have a known good backup must wait for the issue to be resolved for their instance connectivity to be restored.

The case above shows a silent failure caused by an AWS outage. During API call throttling or an outage, the platform should still try to return meaningful messages over API calls rather than fail silently or, worse, carry out the calls without telling the caller.

Erratic Failures

Erratic failures refer to the case when the delivered service is unpredictable.[2] Four percent of cases report erratic failures. This type of call includes two subtypes: 1) the call was pending for a certain time and then returned to the original state, or 2) the call was successfully executed first but failed eventually.

Symptom: A user tried to start the instance several times. It indicated that the status is pending and it goes back to stop.

Root cause: N/A.

Solution: The AWS engineer returned the user's EBS volume to the available state and believed this would resolve the user's problem.

The first sub-type occurs when a user tries to start an instance, which takes 68% of erratic failures. One example is shown above, which was due to the EBS volume issue.

Symptom: A user associated an elastic IP with an instance and could SSH into the instance with the elastic IP. After a few minutes, the elastic IP was silently disassociated from the instance.

Root cause: An issue with the underlying host.

Solution: The AWS engineer advised that the quickest fix was to stop and then start the instance to relocate to a different host.

As shown above, the second sub-type of erratic failures happens to elastic IP association, which is 32% of erratic failures. Cloud consumers cannot completely trust an API call telling you a particular operation has been successfully completed. But it only happens to some specific operations rather than broadly. Cloud consumers should consider a delayed check of status on small calls independently before proceeding confidently to subsequent operations.

FAULTS (CAUSES OF API FAILURES)

The classification proposed in the research of Avizienis and colleagues[2] includes three categories, including development faults, interaction faults, and hardware faults. In the context of Cloud API, development faults concern bugs inside the API implementation or third-party software residing in the cloud platform, not from consumer applications. Interaction faults refer to the ones caused by misconfigurations by API users. We further classify the interaction faults into parameter misconfiguration, component misconfiguration, and software incompatibility misconfiguration.[3] The distribution of misconfiguration faults in our study largely matches the distribution in the report by Z. Yin and colleagues.[3] We proposed resource faults to replace the hardware faults and include hardware faults as a sub-category to suit the cloud computing setting.

Development Faults

Software bugs are a minor cause of API failures. In the reported faults, 2% are due to bugs and 83% come from API implementation bugs. Other bugs are from third-party software residing in the cloud platform. We do not count bugs inside users' applications. Here are several representatives of API bugs confirmed by the development team:

output format of ec2-describe-resereved-instances is messed up

the volume states in the outputs of ec2-describe-instances and ec2-describe-volumes do not match

as-describe-auto-scaling-groups reports wrong launch configuration name for instances which are in 'Terminating' state

the output of --valid-from, --valid-until swapped in ec2-describe-spot-instances

ec2-describe-instances returns an extra undocumented column after the Kernel ID column

ec2-describe-spot-price-history does not report spot price for Linux m1.large instance type

For most of the API bugs reported by the consumers, cloud vendors fix the reported bugs and release a new API version. However, the correction process normally takes several weeks.

Interaction Faults

Misconfiguration is the most frequent cause (32%) of the reported API failures. We went through the 33% API failures caused by misconfiguration faults and reclassified misconfiguration faults into three types: parameter misconfiguration, component misconfiguration, and software incompatibility misconfiguration.[3]

Parameter Misconfiguration

The most frequent misconfiguration fault type is parameter misconfiguration, which is 75% of all misconfiguration faults. In the category of parameter misconfiguration faults, we further classified the reported cases into three types: legal but undesired (18%), illegal value—environment inconsistency (58%), and illegal value—value inconsistency (24%).

Legal but undesired faults refer to the faults that have "perfectly legal parameters but do not deliver the functionality intended by users." The case below shows a frequently occurring legal but undesired fault. There are a large number of cases in the EC2 forum discussing the API issues due to the misconfiguration of the parameter "--region REGION" regarding the --region option and setting of the EC2_URL environment variable. The API Tool Reference (docs.aws.amazon.com/AWSEC2/latest/CommandLineReference/CLTRG-common-args-api.html) does not mention how to set EC2_URL other than us-east-1, although it's easy to guess.

Symptom: When the user runs ec2-describe-spot-price-history, the user only gets results for us-east.

Root cause: Misconfiguration: the user should specify –region option.

Illegal value faults refer to the faults that have correct parameter format but violated value constraints. Illegal value faults include environment inconsistency faults and value inconsistency faults.[3] Environment inconsistency faults refer to when "some parameter's setting is inconsistent with the system environment." The case below shows an environment inconsistency fault example.

Symptom: The user executed ec2-describe-images and received a message saying "The system cannot find the path specified."

Root cause: Misconfiguration: The user set the EC2_HOME path to a wrong path.

Value inconsistency faults refer to when "some parameter settings violate some relationship constraints with some other parameters." Below shows a value inconsistency fault example.

Symptom: When the user tried to start an instance, the instance went to "pending" status, and then went back to "stopped" status.

Root cause: Misconfiguration: The user had tried to attach an EBS volume to xvdq. However, Windows instances cannot have a block device greater than xvdq, which is configured by AWS.

The symptoms and feedback messages around the above two examples can also be improved by incorporating the obviously known and specific cause.

Component Misconfiguration

Twenty percent of misconfiguration faults are component misconfigurations. These cases occur when a component is missing, and they are mainly about misconfiguration in security group settings. Below gives an example of such type of fault.

Symptom: The user was unable to start the instance and getting the error saying "Instance does not have a volume attached at root."

Root cause: There was an inconsistency between the root device configuration and the changed environment.

Software Incompatibility

Five percent of misconfiguration faults are software incompatibility faults, an example of which is shown below.

Symptom: The user was unable to use ec2-describe-regions.

Root cause: Misconfiguration: The user did not install the right version of standard J2SE.

Users usually are not able to receive meaningful feedback from API failures regarding interaction faults. This mirrors the results from Yin when analyzing configuration errors.[3] When an API failure happens, the returned error message should clearly explain failures and faults and advise how to fix them. When a user tries to do some important operations, to avoid misconfigurations, an alert message could explain the outcomes led by the operation and ask the users to confirm the operation. On the one hand, the idea that developers should provide clear error messages when errors are detected is decades old. On the other hand, as our results and results from Yin show, this idea is not universally adopted. An open issue is how to enforce such a requirement.

Resource Faults

A significant percentage of faults (66%) in our study are related to characteristics of the resource being operated on by the API calls. This is significantly different from traditional operator errors and misconfigurations. API calls are essentially operating on coarse-grained resources (virtual machines, volumes, IP addresses, and load balancers). We classify resource faults into four new categories: idiosyncratic rule faults (2%), transient faults (14%), hardware faults (28%), and resource control faults (56%).

Idiosyncratic Rule Faults

Idiosyncratic rule faults refer to the faults caused by violating the idiosyncratic rules that a resource has. These rules are not transient in nature and not related to simple configurations. Here are a few examples:

Windows instances can only have 16 volumes attached and device names only range up to xvdq.

The IDs for the same virtual machine image are different across regions. When users make API calls in a new region, using the wrong ID often causes a failure.

Without rebooting, a Linux instance may not recycle its mount points. If you have mounted more than 24 devices, an API call for attaching a new volume may fail if you try to use the same device name used before even if it is not empty.

A volume cannot be detached from an instance due to corrupted information around processes still accessing the volume.

The error messages for the above failures are often very unclear. The API/system knows why the failures happen but did not communicate that information back to the user. The error message returned by API should be specific.

Transient Faults

Transient faults usually cause time variability of API calls or temporary inconsistency among different calls requesting the same information. Some representative examples are:

Registered instances fail load balancer health check due to timeout but come back later.
There may be a delay (up to 15 minutes) in propagating the SSL certificate to all the regions when the elastic load balancer is created using the AWS management console.
The instance would still appear in the "running" state for some time after it had been terminated.

Normally, the consumer's reaction to this type of faults is to wait and/or retry. However, the waiting time could be different depending on a specific type of transient faults. For example, waiting for SSL certificate to propagate is about 15 minutes, and it may imply a different type of fault after that. Other waiting time could be based on the 95% percentile of the timing profile of an API call.

For tolerating such faults, there is a large body of work on fault tolerance techniques, self-healing mechanisms, and run-time adaptation. However, API failures in cloud are different in nature. API calls in cloud are often requesting different resources to perform tasks on themselves such as starting/stopping oneself and attaching/detaching a volume to oneself. This is different from requesting a resource to do some application-level computation, which is usually covered by the application code rather than infrastructure API calls. There is a possibility that the requesting process produces failures (API implementation or users' API call code) or, more worth noting, the underlying resources performing the tasks produce failures.

Essentially, calling cloud APIs can be seen as offering work items (jobs) to different cloud resources with different design-time and run-time characteristics. Dealing with failures can be seen as exception handling and job (re)offering mechanisms reacting to failures during an operation achieved through a sequence of API calls. There is a body of literature in the workflow community dealing with such issues.[4-5]

Hardware Faults

A significant portion of failures is caused by hardware-related resource faults within the infrastructure. The most common one is degraded underlying hardware upon which the instances reside. Amazon usually sends an email to the owner of the instance notifying the scheduled date of retirement. The common solution to hardware failures is to move the instance to a new physical machine by stopping and starting the instance.

However, users often experience failures when they try to stop the instance and detach the volume due to the failed or degraded underlying hardware. The AWS support team normally advises them to try force stopping or force detachment more than twice to make the operations work. According to comments in API Tool Reference, both force stopping and force detachment operations may cause serious negative impact, like data loss or a corrupted file system, as the instance will not be able to flush file system caches or file system metadata. After either of these two operations, the user must conduct file system check and repair procedures. This option is not recommended for Windows instances.

Many users even fail with several tries of force stopping or force detachment. The AWS support team has to help users fix the issues on AWS side. Therefore, it would be better for the user to immediately stop all services running on the instance with degraded hardware or at least stop accepting new jobs once they receive the notification email.

Resource Control Faults

One important category of resource-related faults is not due to the resources themselves but the control of the resources. This means the resources themselves are healthy but the control planes for starting/stopping resources, redirecting traffic to appropriate resources, or health checking could be the problem causing an API call to fail. This happens in almost all outages.

Twenty percent of the reported API failures in the EC2 forum are part of the Amazon outage events. After each Amazon outage event, Amazon usually publishes a summary of the event and analyzes the reasons for that. There are several signs to the user when AWS outage events happen, for example, stuck API call, timeout API call, and degraded performance. AWS outages can be initially caused by software bugs, natural disasters, network overload, improper hardware upgrade, and power outage but often lead to resource control faults later causing a wider range of failures.

REMEDIES

We have interleaved our recommendations throughout the early sections. We summarize them in the following Table 2 and Table 3. We developed our own solutions to tolerate resource characteristic and configuration faults and late timing failure during operations on cloud applications, namely, POD framework[4] and an API wrapper of AWS API.[5]

Table 2. Recommended Solutions to Each Failure Type.

Failure Classification	Some Recommendations for Cloud Users
Content Failure: Failed Calls with Error Messages	Cloud users can use better knowledge-based diagnosis and configuration checking tools pinpointing the causes of failures.
Content Failure: Missing Content	Cloud users should consider relying on multiple sources of answers for critical operations.
Content Failure: Wrong Content	Cloud users should wait, retry, or compare the answers from multiple sources.
Content Failure: Unexpected Content	Cloud users should perform semantic checks on returned answers.
Late Timing Failure	Cloud users should consider building their own fail-fast framework, build timing profiles on APIs, and use that to fail fast and seek alternatives.
Halt Failure	Cloud users should consider more active backup and immediately make resources into passive/quiescent state when encountering issues and seek alternative means rather than wait.
Erratic Failure	Cloud users should consider a delayed check of status on small calls independently before proceeding confidently to subsequent operations.

Table 3. Recommended Solutions to Each Fault Type.

Faults Classification	Some Recommendations for Cloud Users
Development Fault	Cloud users should treat the answers of API calls as unreliable, perform semantic checks of reasonableness, or compare with answers of other calls.
Interaction Fault: Parameter Misconfiguration	Tools that checks configuration errors are needed.
Interaction Fault: Component Misconfiguration	Tools that checks configuration errors are needed.
Interaction Fault: Software Incompatibility	Tools that checks configuration errors are needed.
Resource Fault: Idiosyncratic Rules	Cloud users can use a knowledge-based configuration checking tool to help.
Resource Fault: Transient Faults	Cloud users should consider using different wait, retry, and fallback strategies for different types of transient faults.
Resource Fault: Hardware Fault	Cloud users should consider more active backup and immediately make resources into passive/quiescent state when encountering hardware faults.
Resource Fault: Resource Control Fault	Cloud users should avoid using API calls during outage and use alternative means such as over-provisioning and zone-aware requests.

CONCLUSION

In this article, we have performed a comprehensive study on Cloud API issues and classified the failures and faults into different categories implying different remedies for cloud platforms and cloud users. Please note that the failure and fault classifications can be reapplied to different case studies. We have built solutions around a fault-tolerant version of the APIs and better exception handlings. Our general conclusion is that software engineers should treat cloud APIs as prone to various types of failures and faults and should use more defensive programming techniques or external solutions to tolerate them.

ACKNOWLEDGMENTS

This project is supported by PetroChina Innovation Foundation (Grant No. 2012D-5006-0304) and the Fundamental Research Funds for the Central Universities (Grant No. 16CX02047A).

REFERENCES

1. Q Lu et al., “Cloud API Issues: an Empirical Study and Impact,” *Proceedings of 9th ACM SIGSOFT Conference on Quality of Software Architecture*, 2013.
2. A Avizienis et al., “Basic concepts and taxonomy of dependable and secure computing,” *IEEE Transactions on Dependable and Secure Computing*, 2004.
3. Z Yin et al., “An empirical study on configuration errors in commercial and open source systems,” *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, 2011.
4. X Xu et al., “POD-Diagnosis: Error Diagnosis of Sporadic Operations on Cloud Applications,” *The 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2014.
5. Q Lu et al., “Mechanisms and Architectures for Tail-Tolerant System Operations in Cloud,” *USENIX HotCloud 2014*, 2014.

ABOUT THE AUTHORS

Zhongwei Li is an associate professor in the College of Computer and Communication Engineering at China University of Petroleum (East China). Contact him at lizhongwei@upc.edu.cn.

Qinghua Lu is an associate professor in the College of Computer and Communication Engineering at China University of Petroleum (East China). Qinghua Lu is the corresponding author. Contact her at qinghualu@upc.edu.cn.

Liming Zhu is a research scientist in Data61 at the Commonwealth Scientific and Industrial Research Organisation (CSIRO). Contact him at Liming.Zhu@data61.csiro.au.

Xiwei Xu is a research scientist in Data61 at the Commonwealth Scientific and Industrial Research Organisation (CSIRO). Contact her at xiwei.xu@data61.csiro.au.

Yue Liu is a post-graduate student in the College of Computer and Communication Engineering at China University of Petroleum (East China). Contact him at s17070790@s.upc.edu.cn.

Weishan Zhang is a professor in the College of Computer and Communication Engineering at China University of Petroleum (East China). Contact him at zhangws@upc.edu.cn.