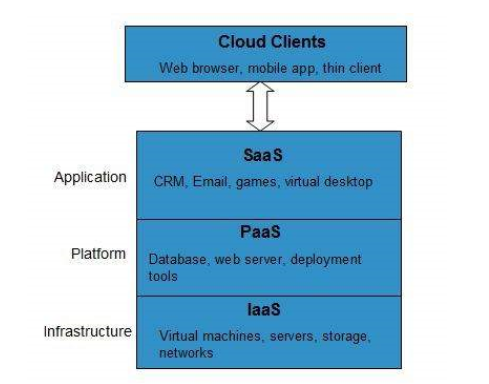
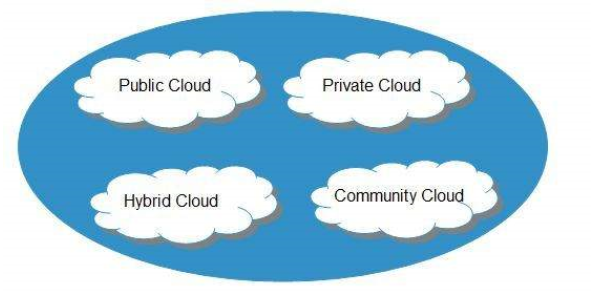
# **Cloud**

### Deployment Models

Deployment models define the type of access to the cloud, i.e., how the cloud is located? Cloud can have any of the four types of access: Public, Private, Hybrid, and Community.



Cloud computing is based on service models. These are categorized into three basic service models which are -

* Infrastructure-as–a-Service (IaaS)
* Platform-as-a-Service (PaaS)
* Software-as-a-Service (SaaS)

**Anything-as-a-Service (XaaS)** is yet another service model, which includes Network-as-a-Service, Business-as-a-Service, Identity-as-a-Service, Database-as-a-Service or Strategy-as-a-Service.

**Infrastructure-as-a-Service (IaaS)**

**IaaS** provides access to fundamental resources such as physical machines, virtual machines, virtual storage, etc.

**Platform-as-a-Service (PaaS)**

**PaaS** provides the runtime environment for applications, development and deployment tools, etc.

**Software-as-a-Service (SaaS)**

**SaaS** model allows to use software applications as a service to end-users.

Benefits

Cloud Computing has numerous advantages. Some of them are listed below -

* One can access applications as utilities, over the Internet.
* One can manipulate and configure the applications online at any time.
* It does not require to install a software to access or manipulate cloud application.
* Cloud Computing offers online development and deployment tools, programming runtime environment through **PaaS model.**
* Cloud resources are available over the network in a manner that provide platform independent access to any type of clients.
* Cloud Computing offers **on-demand self-service.** The resources can be used without interaction with cloud service provider.
* Cloud Computing is highly cost effective because it operates at high efficiency with optimum utilization. It just requires an Internet connection
* Cloud Computing offers load balancing that makes it more reliable.

**Risks**

1. **Security and Privacy**

It is the biggest concern about cloud computing. Since data management and infrastructure management in cloud is provided by third-party, it is always a risk to handover the sensitive information to cloud service providers.

Although the cloud computing vendors ensure highly secured password protected accounts, any sign of security breach may result in loss of customers and businesses.

1. **Lock In**

It is very difficult for the customers to switch from one **Cloud Service Provider (CSP)** to another. It results in dependency on a particular CSP for service.

1. **Isolation Failure**

This risk involves the failure of isolation mechanism that separates storage, memory, and routing between the different tenants.

1. **Management Interface Compromise**

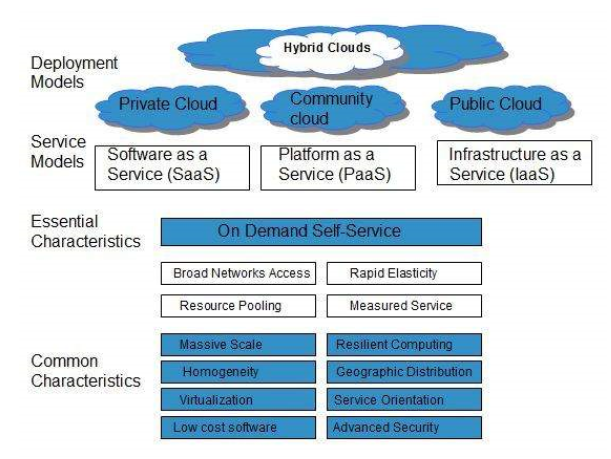
In case of public cloud provider, the customer management interfaces are accessible through the Internet.

1. **Insecure or Incomplete Data Deletion**

It is possible that the data requested for deletion may not get deleted. It happens because either of the following reasons

Extra copies of data are stored but are not available at the time of deletion

Disk that stores data of multiple tenants is destroyed.



# **Amazon Elastic Block Store (EBS)**

Amazon Elastic Block Store (Amazon EBS) provides persistent block level storage volumes for use with Amazon EC2 instances in the AWS Cloud.  Each Amazon EBS volume is automatically replicated within its Availability Zone to protect you from component failure, offering high availability and durability. Amazon EBS volumes offer the consistent and low-latency performance needed to run your workloads. With Amazon EBS, you can scale your usage up or down within minutes – all while paying a low price for only what you provision.

What is Application Resiliency?

Application resilience is the ability of an application to react to problems in one of its components and still provide the best possible service. Resiliency has become more important as organizations continue to rapidly implement software across multi-tier, multiple technology infrastructures.

Why Test for Application Resiliency?

Complex, multiple technology environments have a higher degree of failure and security issues. Testing application resilience provides the ability to assess:

* Non-Conformance to application resiliency standards and best practices
* Security and Privacy issues
* Scalability
* Risk to your business due to application failures

As your organization implements multi-tier systems consisting of several applications and technologies, it is common for vulnerabilities or availability issues between tiers to surface. Automated application resiliency testing offers a dependable method for assessing software while providing measurements to evaluate system performance, architecture standards, and stability as software is rapidly developed or updated.

When bad things happen to good systems

Software solution resiliency refers to the ability of a solution to absorb the impact of a problem in one or more parts of a system, while continuing to provide an acceptable service level to the business. Two significant components in the definition of resiliency here is the problem impact and what service level is considered acceptable when that problem occurs.  
While in an ideal world, a resilient system would be able to deal with any problem in a way that would have no negative impact, but even with proper design and testing it's likely that some user population and requests will be impacted by a failure. If a machine hosting the system (or system component) crashes, any requests that are “in flight” on that machine are moved to another machine - as transparently as possible to the users. More profound in impact is a catastrophic failure in a data center, resulting in all the work that was being processed by that data center to be continued by another data center - again as transparently as possible to the users, although in the event of a catastrophic outage you should be prepared for a significant impact. The goal in this case is to minimize the duration and impact.

How to Develop Resilience Test cases:

The objective of resiliency test cases is to discover how the solution environment will behave when problems occur in the solution environment. For each solution resource identified, a description of a problem to introduce with that resource must be provided in the resiliency test case.  
  
To come up with meaningful resiliency test cases, a good understanding of the following is critical:

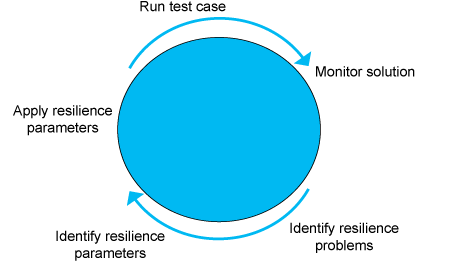
1. **Solution operational model:** The solution resiliency expert must understand the operational model of the solution to a level that enables him to describe the various components and their interactions. A set of sequence or flow diagrams describing, at an operation level, the end-to-end request/response flows is highly recommended. The purpose of these diagrams is to identify, document, and understand the various request/response flows. All resources required by the solution must be identified at the component interaction level.
2. **Solution non-functional requirements:** Although the solution non-functional requirements can comprise a long list of requirements, response time, throughput and availability are the most critical.  
     
   There is no perfect software, things will go wrong. So, you must ask what can realistically go wrong when your software is used in a production environment. For example:

* What happens to the non-functional requirements or service level agreements of a solution flow if one or more resources are unavailable or slow down?
* What happens to the non-functional requirements of the flows if one flow sharing resources with the other flows is having a performance problem?
* Is the availability of a particular resource critical to the business function implemented by a particular solution flow?
* If a resource that is critical to the business function becomes unavailable, and later becomes available, then how would the solution behave? Would the solution still have a problem with that resource?

Questions such as these will drive the development of your resiliency test cases. The use cases that you build should address these areas for each resource:

1. Criticality
2. Problem
   1. Impact
      1. Directly-affected requests
         * Non-functional requirements
      2. Indirectly-affected requests
         * Non-functional requirements
   2. Resiliency test description
      1. Observed solution behavior
      2. Resulting problems (if any)
         * Change to fix resulting problems

| **ID** | **Flow** | **Non-functional requirements** |
| --- | --- | --- |
| 1 | User > FW > WS > AS > DB | RT = 0.5 s TPT = 1000/hour Availability = critical |
| 2 | User > WS > AS > DB | RT = 1 s TPT = 100/hour Availability = non-critical |
| 3 | User > FW > WS > AS > DB | RT = 5 s TPT = 200/hour Availability = critical |



**Conclusion**

For mission-critical applications, it is imperative that resiliency testing is performed before the applications are put in production. If not tested and optimized for resiliency, the consequences can potentially be severe, impacting customer satisfaction levels, and perhaps even resulting in business loss or legal repercussions. The cost and challenge of up-front resiliency testing will almost always outweigh the decision to disregard these risks

Clusters:

For example, you need to ensure that each product listed in the flow is clustered so that if one member of a cluster fails, the other cluster members should be able to assume the work of the failed member. Further, if one member of a cluster fails, what will happen to the non-functional requirements? Will throughputs or response times suffer to the extent tha they will be unacceptable? The solution should employ clusters at every level in a way such that if a cluster member fails, the non-functional requirements are still acceptable.

Story behind Chaos Monkey:

**The Chaos Monkey’s job is to randomly kill instances and services within our architecture.** If we aren’t constantly testing our ability to succeed despite failure, then it isn’t likely to work when it matters most – in the event of an unexpected outage.

At [Stack Exchange](http://stackexchange.com/), Netflix Engineers struggled for months with a bizarre problem. **Every few days, one of the servers in the** [**Oregon web farm**](http://blog.stackoverflow.com/2010/01/stack-overflow-network-configuration/) **would simply stop responding to all external network requests.** No reason, no rationale, and no recovery except for a slow, excruciating shutdown sequence requiring the server to bluescreen before it would reboot.

They spent months -- literally *months* -- chasing this [problem](http://serverfault.com/questions/104791/windows-server-2008-r2-network-adapter-stops-working-requires-hard-reboot) down, walked the list of everything they could think of to solve it, and then some:

* swapping network ports
* replacing network cables
* a different switch
* multiple versions of the network driver
* tweaking OS and driver level network settings
* simplifying our network configuration and removing [TProxy](http://www.balabit.com/support/community/products/tproxy) for more traditional X-FORWARDED-FOR
* switching virtualization providers
* changing our [TCP/IP host model](http://en.wikipedia.org/wiki/Host_model)
* getting Kernel hotfixes and applying them
* involving high-level vendor support teams

At one point in this saga our team almost came to blows because we were so frustrated. (Well, as close to "blows" as a [remote team](http://www.codinghorror.com/blog/2010/05/on-working-remotely.html) can get over Skype, but you know what I mean.) Can you blame us? Every few days, one of our servers -- no telling which one -- would randomly wink off the network. **The Chaos Monkey strikes again!**

Even in our time of greatest frustration, I realized that there was a positive side to all this:

* Where we had one server performing an essential function, we switched to two.
* If we didn't have a sensible fall-back for something, we created one.
* We removed dependencies all over the place, paring down to the absolute minimum we required to run.
* We implemented workarounds to stay running at all times, even when services we previously considered essential were suddenly no longer available.

Every week that went by, we made our system a tiny bit more redundant, because we had to. Despite the ongoing pain, it became clear that Chaos Monkey was actually doing us a big favor by forcing us to become extremely resilient. Not tomorrow, not someday, not at some indeterminate "we'll get to it eventually" point in the future, but *right now where it hurts*.

When you work with the Chaos Monkey, you quickly learn that everything happens for a reason. Except for those things which happen completely randomly. And that's why, even though it sounds crazy, **the best way to avoid failure is to fail constantly.**

Netflix's Chaos Monkey is "a tool that randomly disables our production instances to make sure we can survive this common type of failure without any customer impact," Netflix explained. "The name comes from the idea of unleashing a wild monkey with a weapon in your data center (or cloud region) to randomly shoot down instances and chew through cables—all the while we continue serving our customers without interruption."

Specifically, the Chaos Monkey randomly terminates virtual machines Netflix operates in Amazon's [Auto Scaling](http://aws.amazon.com/autoscaling/) service. In the past year(2012), Netflix says its Chaos Monkey "has terminated over 65,000 instances running in our production and testing environments. Most of the time nobody notices, but we continue to find surprises caused by Chaos Monkey which allows us to isolate and resolve them so they don't happen again."

The Auto Scaling technology on Amazon's cloud should detect the termination of an instance and automatically configure a new, identical one to replace it. But the Chaos Monkey's random attacks can still suss out problems, like a patch gone wrong or a traffic load balancer that's failing to route requests around offline instances.

Amazon's cloud infrastructure is divided into data center regions (like the East Coast or West Coast), which in turn are divided into availability zones. Customers are more likely to survive Amazon failures if they build systems that can fail over across availability zones or regions. Building across regions is the most expensive option, but also the most resilient, as failures have occurred across multiple availability zones on numerous occasions.

**Latency Monkey** induces artificial delays in our RESTful client-server communication layer to simulate service degradation and measures if upstream services respond appropriately. In addition, by making very large delays, we can simulate a node or even an entire service downtime (and test our ability to survive it) without physically bringing these instances down. This can be particularly useful when testing the fault-tolerance of a new service by simulating the failure of its dependencies, without making these dependencies unavailable to the rest of the system.

**Conformity Monkey** finds instances that don’t adhere to best-practices and shuts them down. For example, we know that if we find instances that don’t belong to an auto-scaling group, that’s trouble waiting to happen. We shut them down to give the service owner the opportunity to re-launch them properly.

**Doctor Monkey** taps into health checks that run on each instance as well as monitors other external signs of health (e.g. CPU load) to detect unhealthy instances. Once unhealthy instances are detected, they are removed from service and after giving the service owners time to root-cause the problem, are eventually terminated.

**Janitor Monkey** ensures that our cloud environment is running free of clutter and waste. It searches for unused resources and disposes of them.

**Security Monkey** is an extension of Conformity Monkey. It finds security violations or vulnerabilities, such as improperly configured AWS security groups, and terminates the offending instances. It also ensures that all our SSL and DRM certificates are valid and are not coming up for renewal.

**10-18 Monkey** (short for Localization-Internationalization, or l10n-i18n) detects configuration and run time problems in instances serving customers in multiple geographic regions, using different languages and character sets.

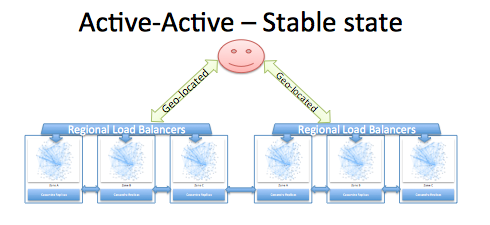
**Chaos Gorilla** is similar to Chaos Monkey, but simulates an outage of an entire Amazon availability zone. We want to verify that our services automatically re-balance to the functional availability zones without user-visible impact or manual intervention.

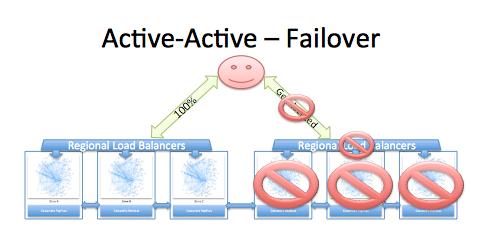
### Active-Active Overview

In a nutshell, Active-Active solution gets all the services on the user call path deployed across multiple AWS Regions - in this case US-East-1 in Virginia and US-West-2 in Oregon.  In order to do so, several requirements must be satisfied

* Services must be stateless - all data / state replication needs to handled in data tier.
* They must access any resource locally in-Region.  This includes resources like S3, SQS, etc.  This means several applications that are publishing data into an S3 bucket, now have to publish the same data into multiple regional S3 buckets.
* there should not be any cross-regional calls on user’s call path.  Data replication should be asynchronous.

In a normal state of operation, users would be geo-DNS routed to the closest AWS Region, with a rough split of 50/50%.  In the event of any significant region-wide outage, we have tools to override geo-DNS and direct all of users traffic to a healthy Region.





Netflix defines Chaos Engineering as the “discipline of experimenting on a distributed system in order to build confidence in the system’s capability to withstand turbulent conditions in production.”

4 principles of Chaos Engineering, according to Netflix, are:

**Build a Hypothesis around Steady State Behavior**

Focus on the measurable output of a system, rather than internal attributes of the system.  Measurements of that output over a short period of time constitute a proxy for the system’s steady state.  The overall system’s throughput, error rates, latency percentiles, etc. could all be metrics of interest representing steady state behavior.  By focusing on systemic behavior patterns during experiments, Chaos verifies that the system does work, rather than trying to validate how it works.

**Vary Real-world Events**

Chaos variables reflect real-world events.  Prioritize events either by potential impact or estimated frequency.  Consider events that correspond to hardware failures like servers dying, software failures like malformed responses, and non-failure events like a spike in traffic or a scaling event.  Any event capable of disrupting steady state is a potential variable in a Chaos experiment.

**Run Experiments in Production**

Systems behave differently depending on environment and traffic patterns.  Since the behavior of utilization can change at any time, sampling real traffic is the only way to reliably capture the request path.  To guarantee both authenticity of the way in which the system is exercised and relevance to the current deployed system, Chaos strongly prefers to experiment directly on production traffic.

**Automate Experiments to Run Continuously**

Running experiments manually is labor-intensive and ultimately unsustainable.  Automate experiments and run them continuously. Chaos Engineering builds automation into the system to drive both orchestration and analysis.

In short, Netflix suggests the following practical steps:

1. Defining what is the normal behavior of a system, its “steady state”
2. Build a control system and an experimental one
3. Start forcing disruptions on the experimental system, simulating real life events such as server crashes, HDD malfunctioning, network failures, etc.
4. Compare the steady state of the experimental system against the control one. The less it deviates from normal, the more confidence one has in the resilience of his system. If problems appear during these experiments, one can learn from what happens and take appropriate measures.

The Plan for Chaos Engineering:

**Establish Virtuous Chaos Cycles**

A common industry practice around outages are blameless post-mortems, a discipline we practice along with action items to prevent recurrence.  In parallel with resilience patches and work to prevent recurrence, we also want to build new chaos tools to regularly and systematically test resilience to detect regressions or new conditions.

Regression Testing in Software Testing is a well understood discipline, chaos testing for regression in distributed systems at scale presents a unique challenge.  We aspire to make chaos testing as well an understood discipline in production systems as other disciplines in software development.

**Increase use of Reliability Design Patterns**

In distributed environments there’s a challenge in both creating reliability design patterns and integrating them in a consistent manner to handle failure.  When an outage or new failure mode surfaces it may start in a single service, but all services may be susceptible to the same failure mode.  Post-mortems will lead to immediate action items for a particular involved service but do not always lead to improvement for other loosely coupled services.  Eventually other susceptible services become impacted by a failure condition that may have previously surfaced. [Hystrix](http://techblog.netflix.com/2012/11/hystrix.html) is a fantastic example of a reliability design pattern that helps to create consistency in our micro-services ecosystem.

**Anticipate Future Failure Modes**

Ideally distributed systems are designed to be so robust and fault-tolerant that they never fail. We must anticipate failure modes, determine ways to inject these conditions in a controlled manner and evolve our reliability design patterns.  Anticipating such events requires creativity and deep understanding of distributed systems; two of the most critical characteristics of Chaos Engineers.

**Resilience Vs Disaster recovery**

Further, there is distinction between “resilience” and disaster recovery. Resilience implies a broader context and a more proactive approach to creating an environment that can absorb the realization of any number of risks and still keep serving the business. Disaster Recovery implies a reactive approach to being able to recover after the fact.

Reference Links:

<http://www.ibm.com/developerworks/websphere/techjournal/1407_col_nasser/1407_col_nasser.html>

<https://blog.codinghorror.com/working-with-the-chaos-monkey/>

<http://techblog.netflix.com/2013/12/active-active-for-multi-regional.html>

<http://www.peak10.com/clustering-in-the-cloud/>

<https://www.infoq.com/news/2015/09/netflix-chaos-engineering>

<http://www.datacenterknowledge.com/archives/2013/10/17/chaos-kong-is-coming-a-look-at-the-global-cloud-and-cdn-powering-netflix/>

<http://techblog.netflix.com/2012/07/chaos-monkey-released-into-wild.html>

<https://blog.codinghorror.com/working-with-the-chaos-monkey/>

<http://principlesofchaos.org/>

<http://techblog.netflix.com/2014/10/fit-failure-injection-testing.html>

<http://techblog.netflix.com/2015/08/from-chaos-to-control-testing.html>

http://www.drj.com/articles/online-exclusive/continuous-application-availability-strategy-for-business-resiliency.html

Slide 5:

**Getting there: 8 best practices for better resilience**To begin developing a strategy of resilience…

* + Identify what will make your business resilient, including organizational and technological factors as outlined above
  + Determine how to measure those factors so you can build appropriate metrics into your key performance indicators (KPIs)
  + Get proactive: set up extensive real-time monitoring and alert systems to measure performance and security stance, and to flag issues ASAP
  + Establish procedures for evaluating technology innovations to determine whether your business should adopt them and how
  + Base decisions on reliable independent data
  + Build robust technology alliances that protect your confidential data and enable you to collaborate effectively
  + Practice for disruption with regular and realistic testing of business continuity plans and don’t forget that specific recovery plans are less useful and dynamic than the ability to create impromptu plans in response to unanticipated situations

Help employees during disruptions, because without a resilient workforce you cannot expect to sustain a resilient business

The benefits of resilience are clear and enable organizations to:

* Adapt/improvise successfully to unforeseen and disruptive changing environments
* Gain a competitive edge by identifying gaps and taking advantage of  opportunities
* Be more agile and innovative by learning from trends
* reduce costs and increase efficiency by avoiding potential pitfalls
* Obtain a better understanding of risks and opportunities
* Preserve and improve their reputation by being seen as vigilant and  robust
* Engender trust amongst external clients and internally amongst staff
* Cultivate a culture of shared purpose and values

For a distributed software solution to be **resilient**, all prerequisite layers supporting this solution must be resilient. These layers are typically the networks, firewalls, load balancers, network switches, operating systems, installed software (such as IBM WebSphere Application Server), and the hosted software solution that might be distributed over a large number of hosts.

Slide 6: **https://www.infoq.com/news/2015/09/netflix-chaos-engineering**

Core idea of Chaos Engineering is to [proactively inject failures](http://techblog.netflix.com/2014/09/introducing-chaos-engineering.html) in order to be prepared when disaster strikes.

*Chaos Engineering is the discipline of experimenting on a distributed system in order to build confidence in the system’s capability to withstand turbulent conditions in production.  
N*etflix affirms the need for finding weaknesses in a production system before they manifest in undesired ways by observing the system’s behaviour in controlled experiments. They outline a number of possible systemic weaknesses: “improper fall-back settings when a service is unavailable; retry storms from improperly tuned timeouts; outages when a downstream dependency receives too much traffic; cascading failures when a single point of failure crashes, etc..”

**Build a Hypothesis around Steady State Behaviour**

Focus on the measurable output of a system, rather than internal attributes of the system.  Measurements of that output over a short period of time constitute a proxy for the system’s steady state.  The overall system’s throughput, error rates, latency percentiles, etc. could all be metrics of interest representing steady state behaviour.  By focusing on systemic behaviour patterns during experiments, Chaos verifies that the system does work, rather than trying to validate how it works.

**Vary Real-world Events**

Chaos variables reflect real-world events.  Prioritize events either by potential impact or estimated frequency.  Consider events that correspond to hardware failures like servers dying, software failures like malformed responses, and non-failure events like a spike in traffic or a scaling event.  Any event capable of disrupting steady state is a potential variable in a Chaos experiment.

**Run Experiments in Production**

Systems behave differently depending on environment and traffic patterns.  Since the behaviour of utilization can change at any time, sampling real traffic is the only way to reliably capture the request path.  To guarantee both authenticity of the way in which the system is exercised and relevance to the current deployed system, Chaos strongly prefers to experiment directly on production traffic.

**Automate Experiments to Run Continuously**

Running experiments manually is labor-intensive and ultimately unsustainable.  Automate experiments and run them continuously. Chaos Engineering builds automation into the system to drive both orchestration and analysis.

In short, Netflix suggests the following practical steps:  
**> Defining what is the normal behaviour of a system, its “steady state”  
> Build a control system and an experimental one  
> Start forcing disruptions on the experimental system, simulating real life events such as server crashes, HDD malfunctioning, network failures, etc.  
> Compare the steady state of the experimental system against the control one. The less it deviates from normal, the more confidence one has in the resilience of his system. If problems appear during these experiments, one can learn from what happens and take appropriate measures.**

**Latency Monkey:**

**Latency Monkey** induces artificial delays in our RESTful client-server communication layer to simulate service degradation and measures if upstream services respond appropriately. In addition, by making very large delays, we can simulate a node or even an entire service downtime (and test our ability to survive it) without physically bringing these instances down. This can be particularly useful when testing the fault-tolerance of a new service by simulating the failure of its dependencies, without making these dependencies unavailable to the rest of the system.

This monkey uses the traffic shaping API to introduce latency (1 second +- 50%) to all network packets. This simulates degradation of the EC2 network.  
This monkey uses the traffic shaping API to drop a fraction of all network packets. This simulates degradation of the EC2 network.

**Conformity Monkey:**

Conformity Monkey determines whether an instance is nonconforming by applying a set of rules on it. If any of the rules determines that the instance is not conforming, the monkey sends an email notification to the owner of the instance. We provide a collection of conformity rules in the open sourced version that are currently used at Netflix and believed general enough to be used by most users. The design of Conformity Monkey also makes it simple to customize rules or to add new ones.

## Why Run Conformity Monkey?

Cloud computing makes it much easier to launch new applications or start new instances. At Netflix, engineers can easily launch a new application in Asgard with a few clicks. With this freedom there are sometimes consequences where launched applications or instances may not follow some best practices, maybe the engineer did not know of the best practice or simply forgot about it. For example, some required security groups may be missing from instances and can cause security gaps. Or perhaps a health check url is not defined for instances in Eureka which would result in automatic failure detection and failover being disabled.

## How Conformity Monkey Works?

Conformity Monkey works in two stages: mark and notify. First, Conformity Monkey loops through all autoscaling groups in your cloud and applies the specified set of conformity rules to the instances in each group. If any conformity rule determines an instance as not conforming, the autoscaling group is marked as nonconforming and the instances that break the rule are recorded. Every autoscaling group is associated with an owner email, which can be obtained from an internal system, or can be set in a configuration file. The simplest way is using a default email address, e.g. your team's email list for all the autoscaling groups. Conformity Monkey sends email notification about the nonconforming groups to the owner, with the details of the broken conformity rule and the instances that failed the conformity check. The application owners can then take necessary actions to fix the failed instances or to exclude the conformity rule if they believe the conformity check is not necessary for the application. We allow you to set different frequency for conformity check and notification. For example, at Netflix, conformity check is performed every hour, and notification is only sent once per day at noon time. This is for reducing the number of emails people receive about the same conformity warning. The real-time result of conformity check for every auto scaling group is shown in a separate UI.

**Security Monkey: Slide 15**

Security Monkey monitors policy changes and alerts on insecure configurations in an AWS account. While Security Monkey’s main purpose is security, it also proves a useful tool for tracking down potential problems as it is essentially a change tracking system.

At Netflix, responsibility for delivering the streaming service is distributed and the environment is constantly changing. Code is deployed thousands of times a day, and cloud configuration parameters are modified just as frequently. To understand and manage the risk associated with this velocity, the security team needs to understand how things are changing and how these changes impact our security posture.

Netflix delivers its service primarily out of Amazon Web Services’ (AWS) public cloud, and while AWS provides excellent visibility of systems and configurations, it has limited capabilities in terms of change tracking and evaluation. To address these limitations, we created Security Monkey - the member of the [Simian Army](http://techblog.netflix.com/2011/07/netflix-simian-army.html) responsible for tracking and evaluating security-related changes and configurations in our AWS environments.

[Security Monkey](http://techblog.netflix.com/2014/06/announcing-security-monkey-aws-security.html), it’s a tool for monitoring and analysing the security of its systems.

Security Monkey monitors and logs changes to configuration changes across any number of Amazon accounts, notifies a user list of users about those changes, and checks the new configurations for common security problems, such as accidentally exposing a server to the whole internet.

**Doctor Monkey** taps into health checks that run on each instance as well as monitors other external signs of health (e.g. CPU load) to detect unhealthy instances. Once unhealthy instances are detected, they are removed from service and after giving the service owner’s time to root-cause the problem, are eventually terminated.

**Janitor Monkey : Slide 16**

<http://techblog.netflix.com/2013/01/janitor-monkey-keeping-cloud-tidy-and.html>

One of the great advantages of moving from a private datacenter into the cloud is that you have quick and easy access to nearly limitless new resources. Innovation and experimentation friction is greatly reduced: to push out a new application release you can quickly build up a new cluster, to get more storage just attach a new volume, to backup your data just make a snapshot, to test out a new idea just create new instances and get to work. The downside of this flexbility is that it is pretty easy to lose track of the cloud resources that are no longer needed or used. Perhaps you forgot to delete the cluster with the previous version of your application, or forgot to destroy the volume when you no longer needed the extra disk. Taking snapshots is great for backups, but do you really need them from 12 months ago? It's not just forgetfulness that can cause problems. API and network errors can cause your request to delete an unused volume to get lost.   
  
At Netflix, when we analyzed our Amazon Web Services (AWS) usage, we found a lot of unused resources and we needed a solution to rectify this problem. Diligent engineers can manualy delete unused resources via [Asgard](http://techblog.netflix.com/2012/06/asgard-web-based-cloud-management-and.html) but we needed a way to automatically detect and clean them up. Our solution was Janitor Monkey.   
  
We have written about our [Simian Army](http://techblog.netflix.com/2011/07/netflix-simian-army.html) in the past and we are now proud to announce that the source code for the new member of our simian army, Janitor Monkey, [**is now open and available to the public**](https://github.com/Netflix/simianarmy).

## Resource Types and Rules

Four types of AWS resources are currently managed by Janitor Monkey: Instances, EBS Volumes, EBS Volume Snapshots, and Auto Scaling Groups. Each of these resource types has its own rules to mark unused resources. For example, an EBS volume is marked as a cleanup candidate if it has not been attached to any instance for 30 days. Another example is that an instance will be cleaned by Janitor Monkey if it is not in any auto scaling group for over 3 days since we know these are experimentation instances -- all others must be in auto scaling groups. The number of retention days in these rules is configurable so the rules can be easily customized to fit your business requirements. We plan to make Janitor Monkey support more resource types in the future, such as launch configurations, security groups, and AMIs. The design of Janitor Monkey makes adding new resource types easy.

## Auditing, Logging, and Costs

Janitor Monkey events are logged in an Amazon SimpleDB table by default. You can easily check the SimpleDB records to find out what Janitor Monkey has done. The resources managed by Janitor Monkey are also stored in SimpleDB. At Netflix we have a UI for managing the Janitor Monkey resources and we have plans to open source it in the future as well.

**Howler Monkey Slide 17**

**AWS Limitations  
To request a limit increase**

1. Open the [AWS Support Center](https://console.aws.amazon.com/support/home#/) page, sign in, if necessary, and then choose **Create Case**.
2. Under **Regarding**, choose **Service Limit Increase**.
3. Under **Limit Type**, choose the type of limit to increase, fill in the necessary fields in the form, and then choose your preferred method of contact.

**Default Limits**

* [Amazon API Gateway Limits](http://docs.aws.amazon.com/general/latest/gr/aws_service_limits.html#limits_apigateway)
* [AWS Application Discovery Service Limits](http://docs.aws.amazon.com/general/latest/gr/aws_service_limits.html#limits_appdiscserve)
* [Amazon AppStream Limits](http://docs.aws.amazon.com/general/latest/gr/aws_service_limits.html#limits_appstream)
* [Application Auto Scaling Limits](http://docs.aws.amazon.com/general/latest/gr/aws_service_limits.html#limits_as-app)
* [Auto Scaling Limits](http://docs.aws.amazon.com/general/latest/gr/aws_service_limits.html#limits_autoscaling)
* [AWS Certificate Manager Limits](http://docs.aws.amazon.com/general/latest/gr/aws_service_limits.html#limits_acm)
* [AWS CloudFormation Limits](http://docs.aws.amazon.com/general/latest/gr/aws_service_limits.html#limits_cloudformation)
* [Amazon CloudFront Limits](http://docs.aws.amazon.com/general/latest/gr/aws_service_limits.html#limits_cloudfront)
* [AWS CloudHSM Limits](http://docs.aws.amazon.com/general/latest/gr/aws_service_limits.html#limits_cloudhsm)
* [Amazon CloudSearch Limits](http://docs.aws.amazon.com/general/latest/gr/aws_service_limits.html#limits_cloudsearch)
* [AWS CloudTrail Limits](http://docs.aws.amazon.com/general/latest/gr/aws_service_limits.html#limits-cloudtrail)
* [Amazon CloudWatch Limits](http://docs.aws.amazon.com/general/latest/gr/aws_service_limits.html#limits_cloudwatch)
* [Amazon CloudWatch Events Limits](http://docs.aws.amazon.com/general/latest/gr/aws_service_limits.html#limits_cloudwatch_events)
* [Amazon CloudWatch Logs Limits](http://docs.aws.amazon.com/general/latest/gr/aws_service_limits.html#limits_cloudwatch_logs)
* [AWS CodeCommit Limits](http://docs.aws.amazon.com/general/latest/gr/aws_service_limits.html#limits_codecommit)
* [AWS CodeDeploy Limits](http://docs.aws.amazon.com/general/latest/gr/aws_service_limits.html#limits_codedeploy)
* [AWS CodePipeline Limits](http://docs.aws.amazon.com/general/latest/gr/aws_service_limits.html#limits_codepipeline)
* [AWS Data Pipeline Limits](http://docs.aws.amazon.com/general/latest/gr/aws_service_limits.html#limits_datapipeline)
* [Amazon EMR Service Limits](http://docs.aws.amazon.com/general/latest/gr/aws_service_limits.html#limits_emr)
* [AWS Database Migration Service Limits](http://docs.aws.amazon.com/general/latest/gr/aws_service_limits.html#limits_dms)
* [Amazon WorkSpaces Limits](http://docs.aws.amazon.com/general/latest/gr/aws_service_limits.html#limits_workspaces)