ARCHITECTURAL DESIGN ON AWS: 3 COMMONLY MISSED BEST PRACTICES

FANATICAL SUPPORT® FOR AWS

Prepared by:

JERRY HARGROVE
Senior Solutions Architect, Fanatical Support for AWS, Rackspace

KIM KING
Product Marketing Manager, Fanatical Support for AWS, Rackspace

JUNE 2016

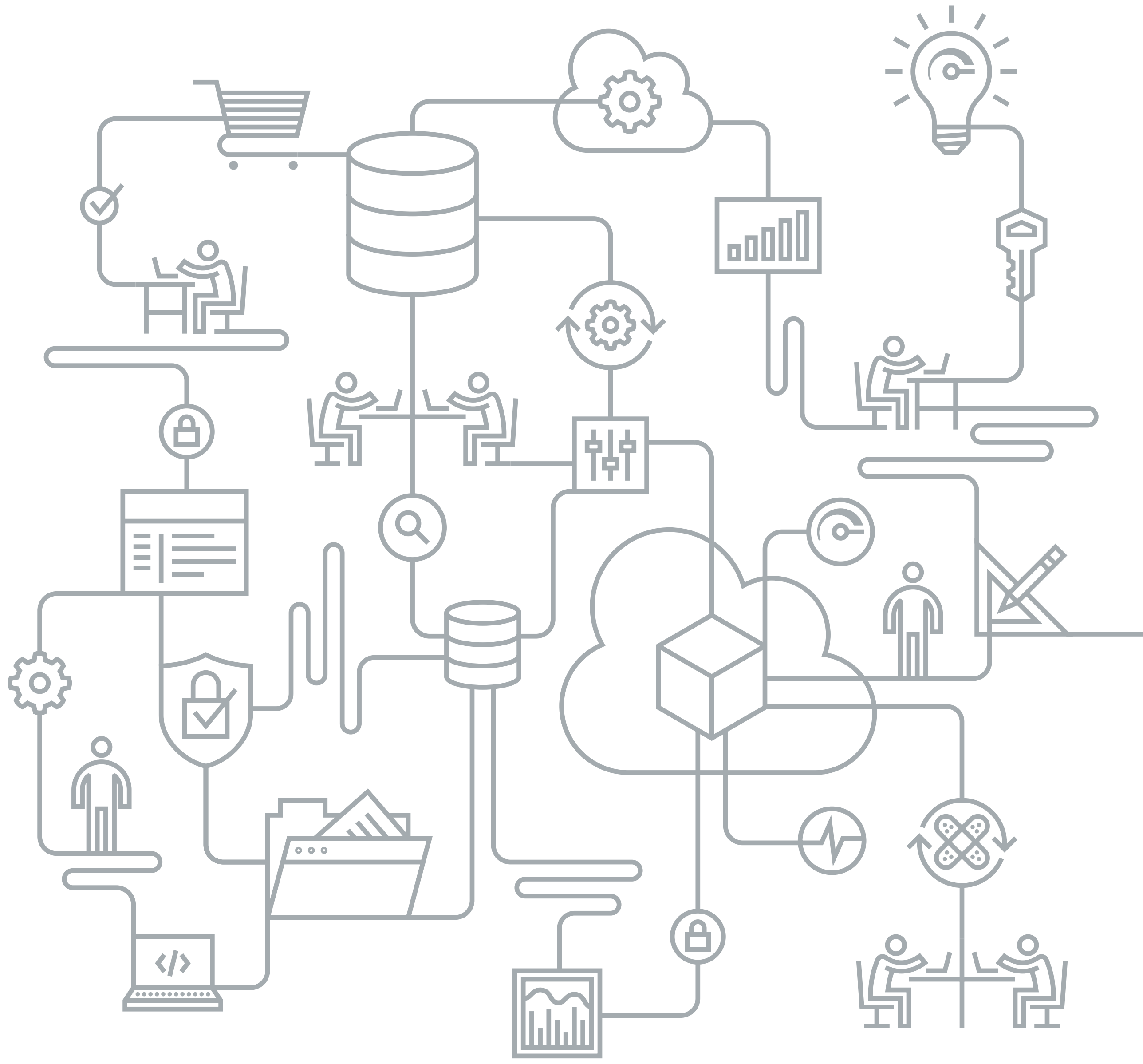


TABLE OF CONTENTS

INTRODUCTION

As businesses make the move to AWS, there are key considerations to be mindful of in order to take full advantage of the power of the AWS platform. In this paper, we will discuss three three commonly overlooked best practices for architectural design on AWS and why adhering to these best practices is a smart business decision.

RACKSPACE RECOMMENDATIONS

Recommendation #1 – Architect for failure

One of the most common mistakes we see when onboarding new customers is that their cloud environments are not architected to respond to failure. Specifically, they have not used automation to back up important data, or included automation to detect and repair unhealthy aspects of their AWS environment without human intervention. Without automation you will need to pay staff members to actively monitor the health of your AWS environment 24x7x365. You are also increasing risk by introducing a single point of failure; the human that is monitoring the health of your environment. Building your AWS environment to automatically fix itself in response to failure dramatically increases the likelihood that your applications will continue to operate even if one or more individual functional component of your system fails, and can prove to be a valuable business decision in terms of both cost reduction and risk avoidance.

***AWS Best Practice Recommendation:** AWS recommends that users "design, implement and deploy for automated recovery from failure."*[1]

To properly architect for failure, assume failures will occur and ensure you have built plans for how these failures will be handled. Use automation wherever possible and consider the implications of failure during each step of the design process.

Automation provides the opportunity to reduce the amount of manual tasks needed to manage cloud workloads and plays a central role in architecting for failure. On AWS, many failures can be dealt with automatically, before the appearance of a problem and without human intervention. Automated management of IT failures in cloud computing has completely changed how server failures are handled, and "all hands on deck" fire drill efforts that adversely impact team productivity can be avoided.

For example, AWS provides services like Relational Database Service (RDS) that can be used to create automated backups of critical data in the event that a server fails.[2] Utilizing this capability to automate backups increases the overall reliability of your system and allows for point-in-time recovery of a database instance.

Architecting for failure also protects businesses from things that are out of their control, such as natural disasters, fires or power outages. AWS users can gain redundancy by distributing applications geographically across multiple Availability Zones and throughout multiple regions. In the event that a data center fails or there is a disaster that affects an AWS region, this helps protect data by running a copy of the application in a far removed geographic location.

To truly understand how a failure will affect a business, you must thoroughly analyze the direct and indirect costs associated with the failure. Faults and failures usually result in data loss or application failure. The extent of the damage depends on the type of failure and the type of business, though Gartner typically cites the average cost of network downtime to be $5,600 per minute.[3] Failures can be detrimental on many fronts. Revenue loss can accrue with every second an application is down, decreased productivity occurs when employees are unable to access applications needed for work or abandon their normal work to scramble to fix the problem, and lost data can be expensive or impossible to replace. Often the most meaningful loss is to an organization's reputation with customers due to applications that fail to work as they should, which becomes especially apparent when a failure results in customer data being lost. Architecting for failure will help you avoid the costly implications of system failures and protect your application data.

***Jerry Hargrove** is a Senior Solutions Architect with over 20 years of architecture and software development experience. He holds multiple AWS certifications, including AWS Solution Architect Professional Certification, AWS Developer Associate Certification and AWS Solution Architect Associate Certification. Jerry is responsible for developing AWS solutions for our customers that are matched to their specific business needs.*

Throughout this paper, Jerry will be chiming in to lend his AWS expertise to our discussion. Here is what he has to say about the concept of architecting for failure:

Architecting for failure includes both failures at the infrastructure level and failures at the application level. Fault tolerance at the infrastructure level can include handling failures of individual Elastic Compute Cloud (EC2)[4] instances, disruptions in service of an Availability Zone, or failure of an RDS instance. Fault tolerance at the application level can include handling downstream service disruptions, database connectivity issues or low resource conditions. Your strategy must include plans to handle failure at both levels, and should provide automated responses to these failures should they occur.

Fault tolerance at the infrastructure level can largely be accomplished by leveraging AWS services and features that are already inherently fault tolerant. For example, choose to use Multi-Availability Zone RDS instance to address service disruption in Availability Zones or failure of the primary RDS instance. Choose to use Elasticache to cache session data to avoid a poor user experience in the case of EC2 instance failure.

Fault tolerance at the application level can also be addressed by leveraging AWS services and features. However, to realize the full benefit of an AWS service's resilience to failure, your application might require some level of change. Many applications still store a user's session information locally and require use of the AWS Elastic Load Balancer (ELB) sticky session feature to maintain user affinity with a particular instance.[5] Session affinity ensures all requests from a user during a session are directed to the same EC2 instance where the session information is stored. This may work in the majority of cases, but fails when the EC2 instance containing the user's session data is no longer available. This can happen if the EC2 instance fails, or as a result of a scaling event which requires the instance to be terminated. When this happens, the user might be forced to log in and could lose work that was stored as part of the session data.

To address this issue, session state should be moved from the instance and stored in Elasticache, Amazon's distributed in-memory data store. Elasticache provides built-in resilience to failure and automates recovery of failed Elasticache instances, but does require changes to the application in order to use the service. For many common application frameworks, this might simply be a configuration change to switch from using a local session cache to a remote cache. For custom applications, application source code may require change to leverage an in-memory cache.

Recommendation #2 – Take advantage of the scalability of AWS services

Another common mistake we see with new customers is application design that does not take advantage of the scalability of the AWS platform. There are several AWS services that are specifically designed to promote the scalability of your AWS environment, such as EC2, ELB and Auto Scaling. Auto Scaling enables compute capacity to scale up or down based on parameters that are defined within your AWS environment.[6] Customers will often run applications without Auto Scaling enabled, which can limit the ease of scalability, affect customer experience within an application and lead to wasted costs associated with applications that are not designed to scale to match demand.

***AWS Best Practice Recommendation:** AWS recommends that users implement intelligent elasticity into their applications wherever possible, and also provides this capability as native in some of their compute services, such as AWS EC2 and AWS Lambda[7].*

With AWS EC2, users can quickly and easily scale compute resources up and down to match application demand. AWS Lambda works in conjunction with EC2 to provide users a way to upload and run their own code without bothering with the administrative pieces necessary to run and scale the code. The recommendation to implement elasticity seems rather intuitive, since scalability and elasticity are two of the most valuable characteristics of running workloads in the cloud, but it is often overlooked when deploying new workloads in the cloud.

Architecting to take advantage of the scalability of the AWS platform provides users the ability to run highly agile applications and also ties back to the concept of architecting for failure. By using Auto Scaling in conjunction with ELB, applications can be architected to automatically distribute incoming traffic across a pool of EC2 instances. Additionally, ELB is able to check the health of EC2 instances and automatically redirect incoming requests to healthy instances until the unhealthy instance has been repaired or replaced. This is a good practice to implement because it enables scalability by helping to ensure that there are always enough healthy EC2 instances running to handle incoming requests and keep applications running smoothly.

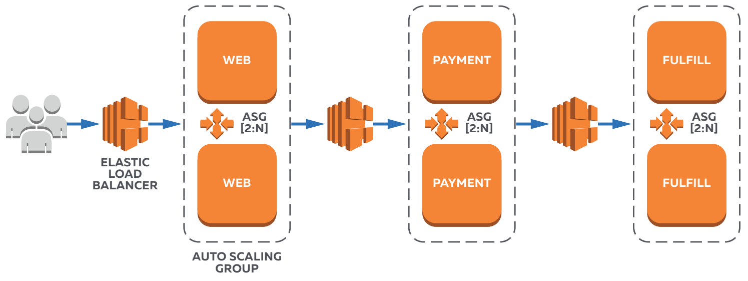
The ability to scale up quickly to meet the needs of increasing customer traffic has obvious benefits, but the importance of being able to quickly scale back down when demand subsides can be overlooked. Applications need to be developed in a particular way to take advantage of scale-down — such as stateless design — and both the app and the underlying infrastructure need to be designed to incorporate this. AWS Auto Scaling allows the user to set usage rules for the deployment and termination of EC2 instances. Users can launch new instances when incoming traffic peaks to ensure a good customer experience, and they can terminate instances when traffic decreases to control costs. This ability to scale symmetrically enables applications to only add the EC2 instances that are needed based on demand for the application, and helps control costs by only launching and paying for compute capacity that will be used.

There are financial and business implications associated with architecting for scalability. If applications are not designed with Auto Scaling to automatically launch a new instance upon failure, EC2 instance failures can result in data loss and application failure when the healthy instances are not available to provide the compute capacity to take incoming requests. Application failures can also result from an inability to scale up to accommodate increases in traffic. Keeping in mind that Gartner estimates the cost of network downtime to be $5,600 per minute, it is easy to see the importance of avoiding these failures from a financial perspective.[8] As discussed previously, application failure provides a poor customer experience and can have devastating effects on the reputation of a business among its customers. Additionally, it makes good business sense to optimize spend and only pay for the resources you will need to use. Auto Scaling enables this by giving businesses the power to easily scale up when demand peaks, and the flexibility to quickly scale down to maximize spend efficiency when peaks subside.

*Here are **Jerry's thoughts** about building for scalability at both the infrastructure and application levels:*

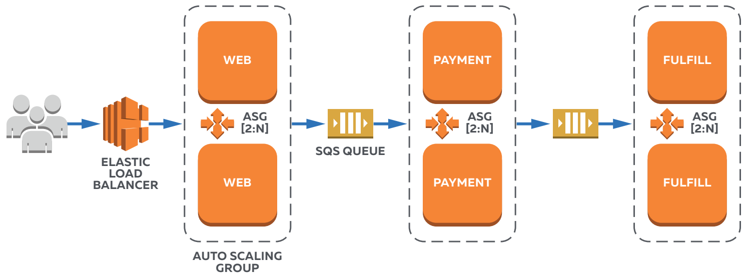
As was the case with fault tolerance, scalability can be viewed from both the application and infrastructure perspectives. At the infrastructure level, inherently scalable services and features such as ELB and Auto Scaling groups can be leveraged to automate horizontal scalability of your applications and services. EC2 also enables easy vertical scaling when your infrastructure requires modification. At the application level, there are other AWS services and features that can be leveraged to increase application elasticity, such as Simple Queue Service (SQS),[9] Simple Notification Service (SNS)[10] and DynamoDB [11].

A chain of dependencies in an application is only as strong as its weakest link, and the scalability of your workload as a whole is only as scalable as the least scalable service it depends on. Take a simple order processing that allows consumers to submit orders and have them fulfilled as an example. Such a system might consist of a web tier for inputting orders, a payment tier for processing credit cards and a fulfillment tier for processing orders. Including common AWS services to ensure fault tolerance and scalability, an architecture might look like this:



In this simple system, orders can only be processed as fast as the slowest subsystem, and the elasticity and scalability of the system is defined by the elasticity and scalability of the least responsive system. For example, if both the web and payment processing tiers can process 100 orders per hour, and the rate of orders exceeds that value, both tiers will be required to scale to meet the increased demand. If either the web tier or payment processing tiers scales slower than the other, the system will not be able to meet the current demand regardless of how fast the web tier can scale. For some period of time, while waiting for the payment tier to scale out, demand will exceed capacity and orders might be lost.

This same issue exists between the payment and fulfillment tiers. To solve this problem and increase the general availability and scalability of this system, we can use queues — specifically AWS SQS queues. Queues allow us to decouple one service from another and allow a buffer to be established between dependent services. This buffer allows upstream services to operate and scale at a different rate than downstream services might be able to, and reduces backstream pressure on the upstream service. Orders can be placed in the queue between services, and the backlog can be managed while the downstream service completes scale-out to handle increased demand. This also greatly enhances the fault tolerance of the web service, as it can now continue to queue up orders in the event that the payment tier suffers some form of outage or is operating at reduced capacity.



A queueing system allows services to scale independently of one another and buffers the effects of fluctuations in demand and capacity. In our order processing system, queues can be placed between each of the services to allow them to operate asynchronously, dampen the effect of scalability mismatches between services and increase the overall scalability of the architecture. The use of SQS also greatly increases the fault tolerance of application architectures by buffering the effects of service outages, as well as in times of reduced capacity.

Recommendation #3 – Create separate AWS accounts for separate environments with one VPC per account

Another common mistake we see in new customer environments is only utilizing a single AWS account for all of the workload environments and segmenting the work environments within the account by using multiple Virtual Private Clouds (VPCs). Using multiple VPCs for this purpose does not provide any additional security benefit, but it does have the potential to complicate operations and can even limit the usage capacity of some AWS services.

> ***AWS Best Practice Recommendation:*** *When setting up AWS accounts, AWS recommends that users "Design your AWS account strategy to maximize security and follow your business and governance requirements".*[12]

AWS offers several design options for how this can be done (Table 1). Rackspace takes this recommendation a step further and suggests that users create a separate account for each deployment environment (development, testing, staging, production) with a single VPC per account.[13] Segmenting the AWS environment this way still meets rigorous security requirements and provides visibility into resource usage — but it eliminates the operational complications that can be associated with segmenting with multiple VPCs.

TABLE 1: AWS RECOMMENDED ACCOUNT DESIGN CONFIGURATIONS

BUSINESS REQUIREMENT	PROPOSED DESIGN	COMMENTS
Centralized security management	Single AWS Account	Centralize information security management and minimize overhead.
Separation of production, development, and testing environments	Three AWS Accounts	Create one AWS account for production services, one for development, and one for testing.
Multiple autonomous departments	Multiple AWS Accounts	Create separate AWS accounts for each autonomous part of the organization. You can assign permissions and policies under each account.
Centralized security management with multiple autonomous independent projects	Multiple AWS Accounts	Create a single AWS account for common project resources (such as DNS services, Active Directory, CMS etc.). Then create separate AWS accounts per project. You can assign permissions and policies under each project account and grant access to resources across accounts.

Each AWS account contains hard and soft limits for the AWS services used within the account[14], such as EC2 instance limits, ELB limits and limits on the number of Simple Storage Service (S3) buckets allowed per account. By creating a separate account for each work environment, users create additional headroom they could need as their account scales up. This "room to grow" reduces the potential risk of hitting the account maximum in any one resource area.

The implications of reaching these service limits are dependent on which AWS service meets the default limit and how important the service is to the ability of the application to function. Consider the following real world example scenario:

An ecommerce site uses a single AWS account that is segmented into three environments: development, testing and production. During a high traffic period for the production environment, a developer launches 15 EC2 test instances. With separate accounts this would not be a problem, but since production and development are housed in the same AWS account, this causes the account to reach the EC2 instance usage limit. Once the account reaches the usage limit, Auto Scaling is prevented from launching new instances needed to match traffic demand on the website, and the website is unable to respond to incoming requests from customers who want to purchase items on their site. With limited compute capacity, the site's response time to requests continues to slow until it is overloaded and the site fails altogether. This results in revenue loss due to inability to access the website. More importantly, it creates bad customer experiences on the site, which could lead to loss of current and future customers.

Segmenting accounts by deployment environment also provides several benefits from a governance perspective by providing increased visibility into the details of resource usage and utilization for each group. When running workloads on AWS it is very important to keep track of user permissions. Creating an account for each different environment makes it easy for management to assign and manage permissions by functional group. This method of segmenting by account also enables management to have more visibility into the spending for each environment, making it easier to optimize spend on IT resources.

Let's get ***Jerry's thoughts*** *on how to address some common scenarios you might encounter when shifting to segment by AWS account:*

If you are currently using the same account for multiple application environments, the use of different accounts for different environments might require development or operational changes.

For example, it is common in continuous integration and development environments to progressively promote an Amazon Machine Image (AMI) between environments as it is being developed and tested. A commit to a source repository might trigger the automated build of an AMI that contained the latest changes. That AMI is then used to validate the changes on a test environment. If the new AMI passes the associated tests, it might be promoted to a QA environment where more rigorous automated testing occurs. Ultimately, this AMI with the latest code changes is promoted to a production environment where it is placed into use. This final promotion can be a manual or automated task. In either case, the addition of per environment accounts requires that this AMI now be shared between accounts before use, a change from the existing process. Sharing the AMI with the AWS Command Line Interface (CLI) is as simple as:

```
aws ec2 modify-image-attribute --image-id <amiId> --launch-
permission "{\"Add\":[{\"UserId\":\"123456789012\"}]}"
```

Another common scenario involves migrating data from a production RDS instance to a test or staging environment, which can be done to allow for more realistic test cases or to reproduce issues that are found in production but not seen in other environments. This requires taking a snapshot of the production RDS instance, followed by creating a new RDS instance in the non-production environment using the production RDS snapshot. The same can occur in a segmented account environment by sharing the RDS snapshot with the non-production account prior to restoring the snapshot to a new non-production RDS instance. Again, this can be achieved with a simple operation using the AWS CLI:

```
aws rds modify-image-attribute --db-snapshot-identifier <snapId>
-- attribute-name "restore" --values-to-add "123456789012"
```

Similar capabilities are available for S3 Buckets, VPC Peering, EBS Snapshots, cloud trail logs and other AWS services and features, each allowing you to share these resources across accounts.

CONCLUSION

Architecting applications on AWS according to best practices can save your business time, money, and preserve the reputational value of your brand. Looking for assistance in implementing these best practices in your AWS workloads? You don't have to go it alone. Fanatical Support for AWS can help provide solutions to these common best practices challenges and many more. Let us help you manage your AWS environment so you can focus on your core business.

With Fanatical Support for AWS, Rackspace blends technology and automation + human experts to deliver ongoing architecture, security, and 24x7x365 operations backed by a team of more than 200 AWS-certified engineers and architects who hold over 300 AWS technical certifications. Two service levels, Navigator and Aviator, provide you the option to choose the Fanatical Support service offering to best meet your business needs. We'd be happy to discuss how Rackspace can help you with AWS — please visit Rackspace.com/AWS or call us at **(844) 858-8886**.

Footnotes

1. Architecting for the Cloud: Best Practices – http://media.amazonwebservices.com/AWS_Cloud_Best_Practices.pdf
2. Learn more about Amazon RDS – https://aws.amazon.com/rds/
3. http://blogs.gartner.com/andrew-lerner/2014/07/16/the-cost-of-downtime/
4. Learn more about Amazon EC2 – https://aws.amazon.com/ec2/
5. Learn more about Amazon Elastic Load Balancing – https://aws.amazon.com/elasticloadbalancing/
6. Learn more about Amazon Auto Scaling – https://aws.amazon.com/autoscaling/details/
7. Learn more about Amazon Lambda – https://aws.amazon.com/lambda/
8. http://blogs.gartner.com/andrew-lerner/2014/07/16/the-cost-of-downtime/
9. Learn more about Amazon SQS – https://aws.amazon.com/sqs/
10. Learn more about Amazon SNS – https://aws.amazon.com/sns/
11. Learn more about Amazon DynamoDB – https://aws.amazon.com/dynamodb/
12. Learn more about AWS Security Best Practices – http://media.amazonwebservices.com/AWS_Security_Best_Practices.pdf
13. Learn more about this Rackspace recommendation – https://manage.rackspace.com/docs/product-guide/recommended_network_configuration/vpc.html
14. Learn more about AWS Service Limits – http://docs.aws.amazon.com/general/latest/gr/aws_service_limits.html

ABOUT RACKSPACE

Rackspace (NYSE: RAX), the #1 managed cloud company, helps businesses tap the power of cloud computing without the challenge and expense of managing complex IT infrastructure and application platforms on their own. Rackspace engineers deliver specialized expertise on top of leading technologies developed by OpenStack®, Microsoft®, VMware® and others, through a results-obsessed service known as Fanatical Support®.

Learn more at www.rackspace.com or call us at **1-800-961-2888**.

JUNE 28, 2016