Building Scalable Applications In the Cloud

Reference Architecture
& Best Practices

RIGHTSCALE®

Brian Adler
Solutions Architect
RightScale, Inc.

Abstract

With the continued expansion of social networks, online media, and various other online communities, website user traffic has become dynamic and oftentimes unpredictable as flash crowd and viral events drive untold numbers of users to sites that may have previously seen little traffic. These situations present many new and unique challenges to application developers and systems architects because they must continually innovate in an attempt to create application and system architectures that can support these previously unseen levels of traffic and system load.

With the growth and acceptance of the cloud as a viable production infrastructure, resources previously out of reach to all but the largest organizations have become available to any group regardless of size. This availability has enabled smaller organizations to develop massively scalable applications, some of which handle tens of millions of daily active users – a feat previously possible in only a small number of enterprise organizations. As application developers and systems architects continue to scale their applications by taking advantage of cloud resources, they are encountering new challenges as they approach previously unseen scalability thresholds.

In this white paper, I will describe in detail three key areas you need to understand to successfully launch a scalable application in the cloud:

1. The Challenges of Building Scalable Applications In the Cloud

I will describe the challenges of building scalable applications in the cloud and recommend architectural designs and techniques that application developers and systems architects can employ at both the application and infrastructure levels to meet these challenges. With these techniques, you can enable applications to scale beyond the thresholds that previously represented your user traffic and system load high-water marks.

2. Best Practices for Building a Reference Architecture

I will also introduce a reference architecture that has been used by many RightScale customers across diverse applications and use cases and, in some situations, been modified to meet their application's specific requirements.

3. Setting Up the Individual Tiers In a Multi-Tiered Architecture

Throughout the majority of this white paper, I will focus on the individual tiers of a multi-tiered architecture and will pay particular attention to the unique characteristics and scalability considerations for each tier of the architecture.

© 2011 RightScale, Inc. All rights reserved.

1

Introduction

The cloud provides the ideal environment for scalable applications because it allows for rapid resource allocation in times of high demand as well as resource deallocation as demand declines. The resources and infrastructure of the cloud can accommodate all phases of an application's lifecycle, providing a single context in which to bring an application from concept to development, then from production to maintenance, and finally, to a gradual end of life.

RightScale has a diverse group of customers who use the RightScale cloud management platform to manage their scalable applications. As a result of our extensive experience in supporting these distinctive environments, we have had the opportunity to gain a unique perspective on all aspects of the application lifecycle — in particular the architectural designs that have allowed applications to flourish as well as the potential pitfalls that should be avoided.

Although the merits of a scalable architecture are readily apparent in environments that may be influenced by viral events, organizations can also realize tremendous benefits by applying a scalable architecture to applications that have more stable and predictable usage patterns. In these environments, you can leverage the scalability the cloud affords to make the most productive use of development and testing time when introducing new features to an application.

It is not uncommon for new features to place unexpected loads on a system when they are introduced. Exhaustive testing of the performance characteristics of these new features before release may be possible, but this often comes with significant cost both in time to market as well as in infrastructure and manpower. By using a scalable architecture, you can absorb sudden increases in processing time due to the addition of new features by scaling to accommodate the increased load they impart on the system until you can isolate and optimize the performance bottlenecks.

And although the flexibility that scalable resources provide should never be used as a substitute for comprehensive functional and load testing, they can ensure a safety net in the event that unexpected application characteristics are exposed during real-world use cases. Additionally, certain leading-edge new features may push the boundaries of traditional architectures with the extra processing demands that they require, rendering testing in on-premise resources impractical. A scalable architecture facilitates the deployment of these features, allows the market to be tested, facilitates the identification of real use cases, enables performance to be characterized under real-world conditions, and allows resource requirements to be improved over time as business needs dictate.

In subsequent sections of this white paper, I will describe the concepts and components of scalable applications as well as the traditional methods used to address the needs presented by these applications. I will contrast these traditional methods with the methods of the scalable cloud resource model and will illustrate the alignment of these methods to the needs of an application in a dynamic environment. I will also describe architectural best practices and explain in detail how to set up each tier of a reference architecture that has been used with great success by many RightScale customers.

In conclusion, I will summarize the findings that we at RightScale have gained through our experience in the scalable application realm. These include our insights on architectural considerations and a presentation of additional mechanisms that are available to enable the deployment and maintenance of a scalable application in the cloud.

© 2011 RightScale, Inc. All rights reserved.

2

Scalable Applications

A scalable application can take many forms, but in essence it is an application and underlying infrastructure that can adapt to dynamically changing conditions to promote the availability and reliability of a service. With the proliferation of online communities and the potential for cross-pollination across these environments, the traffic and load patterns encountered by an application have become unpredictable as the potential for viral or flash crowd events can drive unprecedented traffic levels. This dynamic nature drives the need for a massively scalable solution to enable the availability of web-based applications.

Previously, in the traditional hardware model, there were two approaches you could take with regard to the issue of the unpredictability of site traffic and system load, each of which is illustrated in **Figure 1.**

The first approach was to overprovision, that is, to have enough resources in place to handle any spikes in traffic that may occur. While this enables an application to increase its availability in high-traffic situations, it is not an effective use of resources because a portion (and perhaps the majority) of these resources sits idle during non-peak periods. This inefficiency is illustrated in **Figure 1** by the gap between the blue line representing infrastructure costs (which can be generalized to represent the number of servers in use), and the red line that is an indication of actual user demand for the service provided by the application. The gray arrow illustrates the disparity between the two. This scenario is obviously a costly solution due to the presence of unutilized capacity and therefore generally not a common or recommended approach.

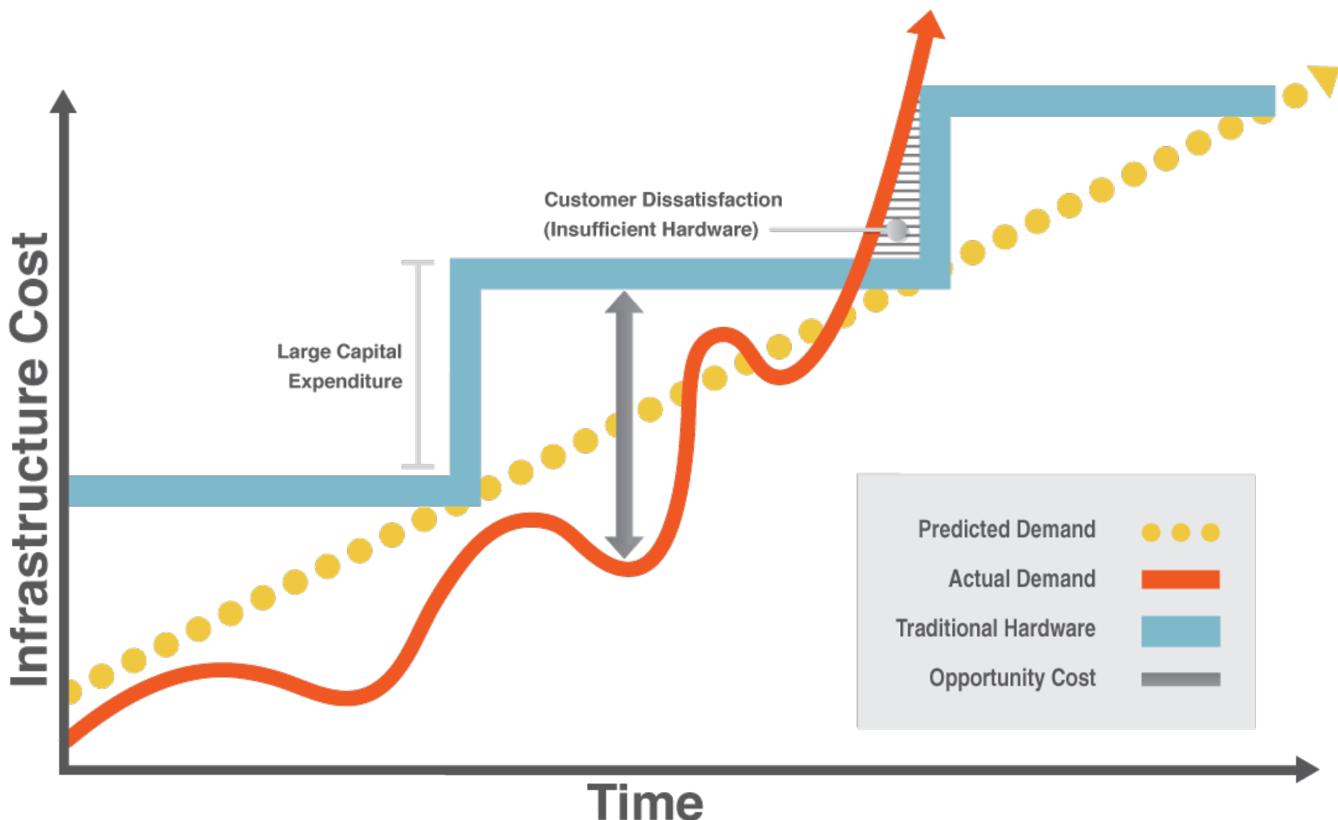


Figure 1 – Traditional Hardware Model

© 2011 RightScale, Inc. All rights reserved.

The second approach in the traditional hardware model is to provision for the typical usage pattern of the application and suffer the consequences of lost traffic when peak demands are encountered. Although this is more cost friendly in times of normal usage, it is costly during traffic spikes because lost traffic typically means lost revenue opportunities. This scenario is illustrated in **Figure 1** by the shaded region under the demand curve represented by the red line and above the available infrastructure capacity represented by the blue line. In this situation the demand exceeds capacity, and, as a result, traffic is lost and/or the application service is unavailable.

Neither of these approaches in the traditional hardware model is ideal, which is why the scalable cloud model is such an excellent fit for this type of dynamic and unpredictable environment. With the scalable cloud model, you can dynamically provision additional resources only when they are needed and then decommission them when they are no longer required. In true utility computing fashion, you incur charges only for the time period in which you use the resources. **Figure 2** illustrates the scalable cloud model for application resources.

In **Figure 2**, the demand curve is identical to that of **Figure 1**, but due to the dynamic provisioning of resources, at no time is there excess capacity in which servers sit idle, nor is there insufficient capacity to accommodate the demand for the application. In the following sections, I will describe the preferred methods and techniques for best implementing the scalable cloud model at all levels of an application's multi-tiered architecture.

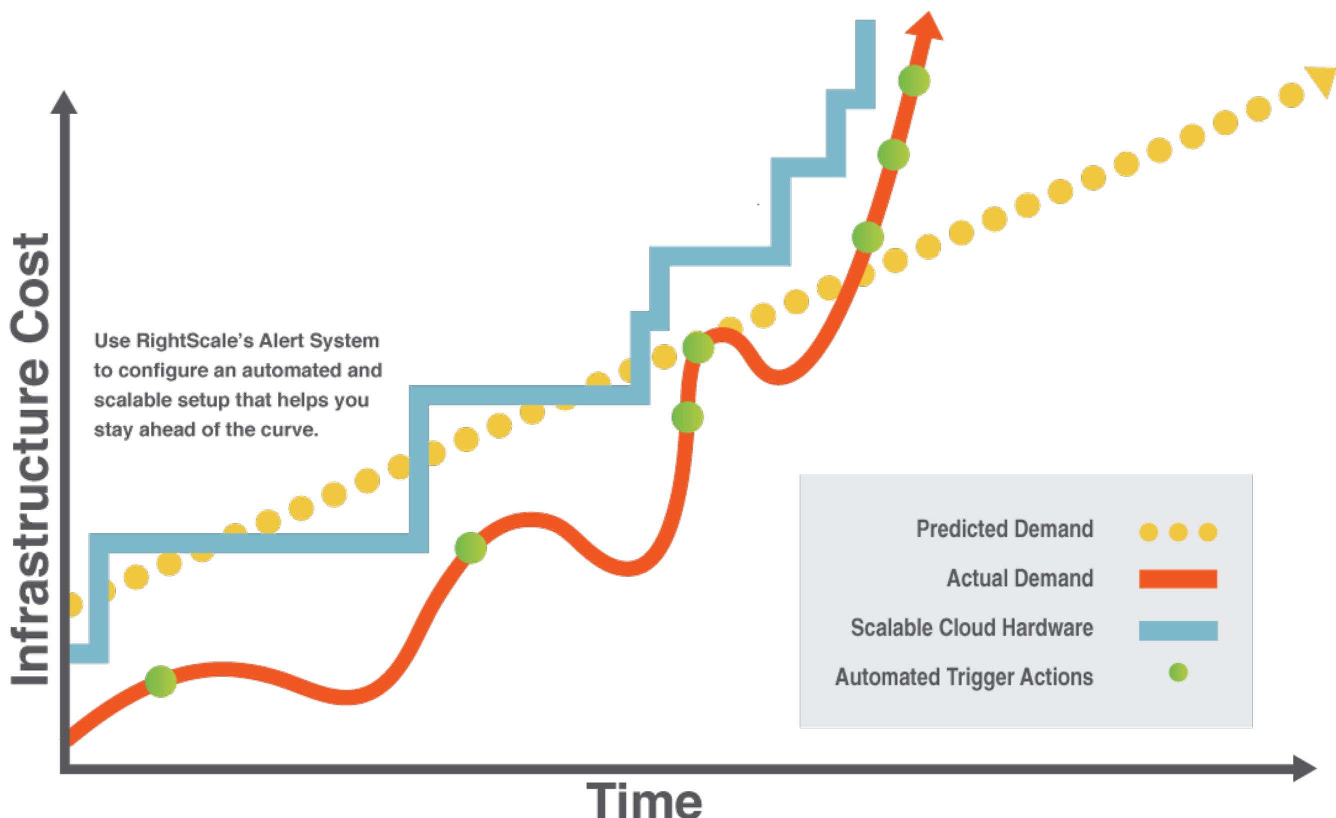


Figure 2 – Scalable Cloud Model

© 2011 RightScale, Inc. All rights reserved.

3

Reference Architecture & Best Practices

RightScale has had the opportunity to work with many organizations in diverse industries with distinctly different use cases. This exposure has enabled us to collect a wealth of knowledge on best practices for creating successful cloud architectures for use with scalable applications. In this section, I will describe a reference architecture for a scalable web application and outline the distinct tiers of this typical architecture, as well as share tips and techniques for optimizing the functionality provided by each of these tiers.

Figure 3 illustrates a reference architecture commonly used by scalable applications that incorporates the best practices gathered through RightScale's successful implementations for our diverse customer base. You'll see that this architecture looks much like the classic three-tier web application architecture with the addition of a caching tier between the application servers and the database. However, there are some subtle (and some less subtle) modifications and enhancements that you can make to the architecture to allow it to best handle the unique traffic and load patterns experienced by scalable applications.

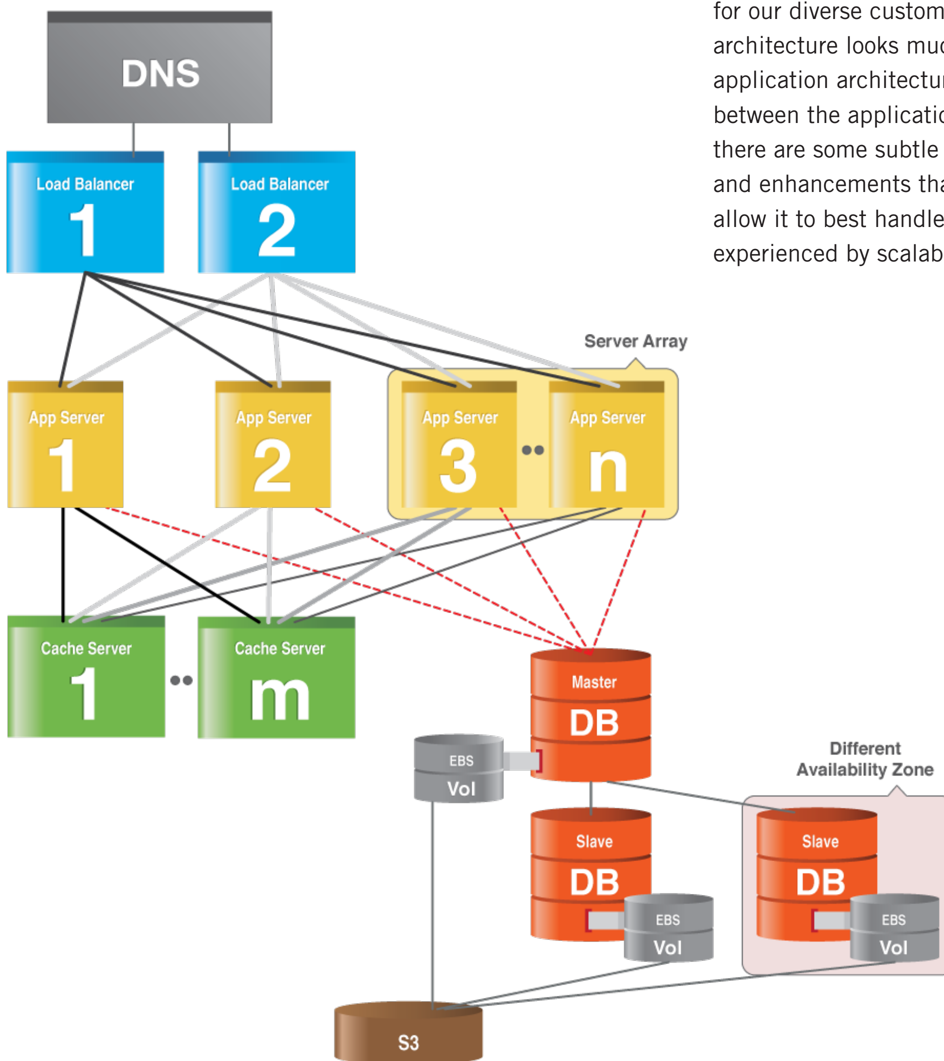


Figure 3 – Scalable Web Application Reference Architecture

© 2011 RightScale, Inc. All rights reserved.

3.1

Load Balancing Tier

The first tier shown in the reference architecture of **Figure 3** is composed of two load balancers, typically running HAProxy. These load balancers are commonly run on the Amazon Web Services (AWS) Elastic Compute Cloud (EC2) m1.large instance types, which provide 2 virtual cores, 7.5GB of memory, and a 64-bit platform. Our extensive testing on this configuration has shown it has the capacity to handle approximately 5,000 requests per second on each load balancer, thus giving a combined total of about 10,000 requests per second for the example shown. If possible, estimate your site's highest traffic rate in terms of requests per second, and divide this value by 5,000, which will give an approximate number of load balancers required to handle the traffic load. (For detailed descriptions and analysis of the load-balancing tests we performed, please see our whitepaper on Load Balancing In the Cloud: Tools, Tips and Techniques at http://www.rightscale.com/info_center/white-papers/load-balancing-in-the-cloud.php)

Regardless of estimated load, we recommend using two front-end load balancers to provide redundancy in the case of a server failure. Additionally, we recommend that these load balancers be placed in different availability zones to increase the reliability and availability of the application. (An availability zone can be thought of as a separate data center, with its own power, network connectivity, etc.) For the vast majority of new deployments, two load balancers are sufficient in an application's early lifecycle phases. For example, a current RightScale customer in the social gaming space has handled more than 30 million daily active users, which would not be possible with just two load balancers. However, the customer's initial deployment was similar to the configuration shown in **Figure 3**, and additional load balancers were added as traffic to the site increased.

Because cost management should always be a consideration, you can initially run load balancers on m1.small instances (1 virtual core, 1.7GB memory, 32-bit platform) as a cost-cutting measure and then migrate them to larger instances as demand increases. In the early phases of an application's lifecycle, you can combine the front-end load balancers and the application servers on the same instance to allow for additional cost savings. Although we do not recommend this for a high-traffic production site, it is often an effective approach when resource demands are low. When site traffic increases and these dual-function servers begin to become overloaded, you can run a script that will remove the application server from the pool of servers handling application requests, thus resulting in the server functioning purely as a load balancer. This application server separation should only be done once the application array has been populated with adequate resources to handle the processing load, and when array minimums and maximums have been set correctly (I will provide more detail on these concepts in section 3.2).

Although auto-scaling front ends is theoretically possible, we do not recommend it due to the complexities of configuring DNS programmatically using a DNS Application Programming Interface. However, if you wish to auto-scale front ends, RightScale provides tools that facilitate adding front-end load balancers when necessary. The tools enable you to manually launch the new load balancer, and all application servers in the deployment can be triggered to automatically register themselves with this new front end.

© 2011 RightScale, Inc. All rights reserved.

3.1 (cont.)

An additional option for implementing load balancing functionality is the Elastic Load Balancer (ELB). The ELB solution is only available within the AWS infrastructure, as opposed to an HAProxy-based solution that is open source software and available on any Linux platform.

An ELB is essentially a load-balancing appliance. As such, your configuration options are limited and you have very restricted visibility into the ELB's inner workings and performance characteristics. However, an ELB is an intrinsically scalable solution in that it will automatically scale to accommodate increased load. The ramp-up time for this additional capacity may be a limiting factor for certain applications, such as environments where rapid traffic increases may occur. (Please see our load-balancing white paper for additional details on ELB: http://www.rightscale.com/info_center/white-papers/load-balancing-in-the-cloud.php)

For any environment in which rapid, large increases in traffic may occur, we always advise that you run load tests against any chosen architectural component to ensure that the needs of the application are met.

© 2011 RightScale, Inc. All rights reserved.

3.1.1

Content Delivery Network

When discussing the user-facing tier of an application, it is worth noting the potential benefits of a content delivery network (CDN) to deliver static content from edge locations. Many user-interactive applications require the initial download of a thin (and sometimes not-so-thin) client to the end user.

This client can reside in the database and/or caching tiers (I'll cover this in more detail in sections 3.3 and 3.4), but this implementation is generally not a recommended best practice for a site with more than light to moderate traffic due to the network throughput demands placed on the application.

Although cost considerations must come into play with any CDN, particularly in the early phases of an application's lifecycle when revenues are lower, the benefits of such a solution can be far-reaching. The user experience can be enhanced by faster downloads from server edge locations that are geographically close to the user (and therefore have lower latency), and the site will obviously benefit from the reduced bandwidth required to service the download request. Differing architectural options are possible, and depending on the site's traffic patterns, CDNs may not be necessary, but you should consider the potential benefits of a CDN through all phases of the application lifecycle.

 © 2011 RightScale, Inc. All rights reserved.

3.2

Application Tier

The second tier of the architecture shown in **Figure 3** consists of the application servers and the associated scalable server array. This tier is typically initially configured with two application servers (in different availability zones) and with alert mechanisms in place to allow automatic scaling (both up and down) of the array based on instance-specific metrics. The most common metrics used for auto-scaling are CPU-idle, free memory, and system load. However, you can use virtually any system metric as an auto-scaling trigger, including application-specific metrics that you can add via custom plug-ins to the RightScale monitoring and v.

When any of the thresholds specified by these metrics are met, the custom alert associated with that metric is initiated, which can result in a scaling up of additional application servers in the case of increased demand, or the decommission of active servers if the load decreases. As mentioned in section 3.1, during the early phases of an application's lifecycle, these application servers may be merged with the front-end load balancers to save on infrastructure costs.

As a best practice, we recommend that application server arrays scale up in a conservative manner and scale down conservatively as well. Accordingly, you should launch additional instances before they are needed when you detect an upward trend in activity.

It is important to determine the time required for a server to become operational after launch and to factor this into the scaling metrics. Similarly, you should only decommission instances when they have been lightly utilized for a pre-determined period of time. Scaling up conservatively (that is, when lower thresholds are met) helps to ensure that resources are continually available to your application, while scaling down conservatively prevents terminating application server instances prematurely and causing an undesirable user experience.

The only disadvantage is that if you take too conservative an approach when scaling up, you will be charged additional server time. But in the true nature of utility computing, if a server was launched unnecessarily, you would incur charges for one hour only (the smallest billing granularity) because the scale-down metrics would terminate the server before the next billable hour began.

You can configure arrays to have both a minimum and a maximum number of instances. If you use the combined load balancer and application server approach, typically you will set this array minimum to zero. Once application traffic has increased and array instances have been launched, this minimum should be increased (and must be increased if the application servers are removed from the front-end instances, since this may result in all application servers being terminated in a scale-down event if the minimum remains at zero). The maximum array size allows an upper bound to be placed on the total number of instances running, and thus places a limit on infrastructure costs. You should determine the optimal instance size for an application server in a scalable array via load testing and performance benchmarking. However, many RightScale customers have found the AWS m1.large instance size to be appropriate for most of their application server needs.

When it makes sense from a user-experience perspective and when it's cost feasible to do so, we advise that you launch application servers in multiple availability zones (within the same region/cloud) to increase reliability and availability. On this point it is important to note that data transfer between EC2 availability zones and regions is charged on a per-gigabyte basis, and latency across these zones and regions may adversely affect an application's performance. Although we do not recommend cross-region deployments due to the cost and latency involved, the EC2 architecture provides reasonably fast connectivity between zones within the same region and at reasonable bandwidth costs. As such, dispersing server instances across all availability zones in a selected region is a recommended best practice with regard to application availability and reliability.

© 2011 RightScale, Inc. All rights reserved.

3.3

Caching Tier

The third tier shown in **Figure 3** is the caching tier, which is typically implemented with memcached. Not all application architectures benefit from a data-caching solution, but the majority of scalable applications will realize increased performance via the use of a distributed cache. For applications that are read-intensive, a caching implementation can provide large performance gains as application processing time and database access is reduced, sometimes dramatically. Write-intensive applications typically do not see as great a benefit, yet there are modifications you can make to the classic caching paradigm that can assist in achieving considerable performance gains in these environments as well.

Most caching solutions (such as memcached) are fairly lightweight in terms of CPU utilization, but heavy (as heavy as the user will allow) on memory usage, so we advise that you use larger instance sizes (in terms of memory) for servers in this tier. In the early phases of an application's lifecycle, the caching footprint (the total size of cached objects) tends to be small, and your instinct may be to use a single instance to provide the cache for the entire application server tier. While this is effective in light to moderate usage situations, we do not advise it for production applications as traffic increases.

A single caching server represents a potential single point of failure for the application cache, and a loss of this instance can result in a severe performance hit to the application and backing database. As such, we recommend that you use multiple instances (distributed across availability zones within the selected region/cloud) in the implementation of the caching tier. An approach we advocate is to approximate the required caching footprint and then to overprovision the required memory by some percentage across multiple instances. This overprovisioning provides a buffer of extra caching capacity as the application's usage increases, and the distribution across multiple instances in multiple availability zones provides reliability and availability while eliminating single points of failure.

Auto-scaling a memcached tier is not a recommended best practice since it typically requires a configuration file update and a restart of the application, but it is a fairly straightforward manual process. When additional memcached servers are added, the hashing algorithms used by the application servers to locate the correct memcached server will potentially map to different caching servers, so there will most likely be an increase in database load as the previously cached objects (and newly requested objects) are distributed across the new larger caching pool.

It is this ability to add caching servers that drives a recommended best practice regarding time-to-live (TTL) values on cached objects: Although TTLs can be set to indefinite values (that is, no timeout), we do not recommend it. You should always set TTLs to expire because the addition of a new caching server may result in a redistribution of the cached objects on an existing server. If a redistributed object never expires from the cache, it will reside in memory indefinitely, thus unnecessarily consuming resources. A restart of the memcached process will clear the cache of any non-expiring objects, but this will also affect all other objects in the cache, and application and database performance will be negatively affected as the cache is repopulated over time.

For applications with a small caching footprint, you can install memcached co-resident with the application on the individual application server instance. While we do not recommend this in a production environment, this implementation can be used as a cost-savings measure in development and test environments since a separate tier of caching servers is not required.

© 2011 RightScale, Inc. All rights reserved.

3.3 (cont.)

A disadvantage of this implementation is that cached information is most likely duplicated across application server nodes, thus multiple database accesses are required to populate the same data object in each of the caches. If a caching solution is used on each application server, each cache should be used only for local cache purposes and not as part of a distributed cache. If these caching servers were consolidated into a distributed cache, then as application servers launch and terminate in the scalable application tier to handle the dynamic traffic load, there would be constant turnover in the distributed cache, which would result in application and database degradation as the cached objects were constantly redistributed.

As I mentioned in reference to content delivery networks in section 3.1.1, scalable applications often require the download of a client to the end user. In sites that see moderate traffic, these clients can reside in the caching tier to prevent unnecessary access to the database for static content. As traffic increases, however, you should constantly re-evaluate the inclusion of content delivery mechanisms to alleviate the bandwidth requirements of your application.

© 2011 RightScale, Inc. All rights reserved.

3.4

Database Tier

The final tier shown in **Figure 3** contains the database/persistent datastore. As with any web-based application, this tier is of critical importance and can be challenging to architect correctly. While there is no "one-size-fits-all" solution when it comes to data storage and management, there are typical categories and types of applications that draw from a common set of architectural components and best practices. In the sections to follow, I will detail many of the options available for database design and implementation, pulling from the knowledge and experience gathered from RightScale's diverse customer base across many industries and use cases.

There are numerous database applications that you can implement in a web application technology stack, but the most common and widespread of the open source packages is MySQL. The architecture shown in **Figure 3** illustrates a recommended best practice for MySQL use in the cloud, specifically the presence of one (or more) slave databases.

Although using cloud-based resources enables application flexibility, the physical inaccessibility of these resources requires additional planning and forethought. While not common, hardware failures in the cloud do occur and need to be planned for accordingly. It is for this reason that we always recommend the use of one or more slaves. If a master database fails, a slave can be quickly promoted to become the new master using pre-configured scripts. And if budget allows, we recommend that you create additional slaves in different availability zones to increase the availability and reliability of the datastore.

You should also carefully choose the disk subsystem configuration used for any database implementation. Instance-attached, or ephemeral, storage can be used for a database, but we strongly advise against this because an unexpected failure or termination of the instance will result in a total loss of the database. And although replicating slaves can assist in mitigating data loss, any transactions that were not yet in transit to the slaves will be lost. Instead, we recommend the use of an AWS Elastic Block Store (EBS) volume (or your cloud's equivalent) or a set of volumes in a striped configuration.

EBS volumes are persistent, so in the event of an instance failure, the EBS volume (and the data contained therein) will not be lost. Additionally, EBS volumes enhance the efficiency in which backups can be made and restores can be performed from those backups.

You have many options with regard to the setup and configuration of EBS volumes for use as the backing data store at the database tier. Although the current size limit on an EBS volume is 1TB, RightScale has automated tools that allow the creation of a RAID 0 (striped) configuration across multiple EBS volumes, thus allowing larger logical volume sizes (for example, four 1TB volumes could be incorporated into a single logical 4TB volume).

To enhance I/O performance, you can incorporate a disk design best practice that involves the use and placement of MySQL's binary logs, or bin logs. We recommended that you write bin logs to a disk other than the volume that contains the database. RightScale has seen increased database performance by writing the bin log data to the instance's ephemeral drive. By physically separating the database I/O from the bin log I/O, the physical disk(s) implementing the database storage can be utilized for the database only, thus improving overall throughput of the database's disk subsystems.

Another practice we recommend is to take periodic snapshots of the database. A successful method used by RightScale is to create an XFS filesystem on the EBS volumes (individual or striped) to facilitate snapshotting of the filesystem. XFS allows the filesystem to be "frozen," during which time the snapshot is performed. You should take snapshots on a scheduled basis (slave snapshots can be performed more frequently since the load on these servers is lower than on the master) and identify times of low I/O load and incorporate them into these backup schedules. Upload these snapshots to a distributed, persistent storage infrastructure (such as S3 or equivalent) for archival, backup, and recovery purposes.

© 2011 RightScale, Inc. All rights reserved.

3.4.1

Database Scaling

For an application to continue to be successful as its lifecycle progresses, it has to be scalable at all levels of the architecture. As more users interact with the application, the resource demands of each tier will continue to increase. In previous sections of this white paper I outlined some of the mechanisms that can be used to scale the front-end load balancers, the application server array, and the caching tier. However, the success or failure of many applications is dependent on a well conceptualized, architected, and implemented database system. And while the ultimate goal of database design would be to allow the automated horizontal scaling of the database tier, the practical implementation of such a solution continues to remain an elusive goal. However, there are design concepts you can incorporate to allow database scaling to varying degrees. In the following sections, I will discuss options for both vertical and horizontal scaling the database tier.

© 2011 RightScale, Inc. All rights reserved.

3.4.1.1

Vertical Scaling

In the early stages of an application's lifecycle, when database load is light, you can often effectively use a small instance size for both the master and slave databases. As load increases, you can migrate the master database to a larger instance size, allowing it to take advantage of additional processing power, I/O throughput, and available memory. Current AWS EC2 instance sizes range from 32-bit platforms with 613MB of memory up to 64-bit instance types with 68.4GB memory.

For database requests that involve complex queries or joins of multiple tables, the additional memory provided by the larger instance types can assist greatly in accelerating the query response. When possible, the working set of a database should be contained in memory because this greatly reduces the disk I/O requirements of the application and can greatly enhance the application's overall performance. Situations may arise in which the CPUs of an instance are greatly underutilized yet the majority of memory is in use. Although this may appear to be a poor use of a powerful (and costly) resource, the performance gains realized by containing the working set entirely in memory can far outweigh the costs incurred by these more expensive instance sizes.

RightScale provides scripts that allow the migration of a database from one instance size to a larger (or smaller) instance with virtually no database downtime. Many of our customers initiate an application deployment with their master and slave databases on small- to medium-size instances and then migrate their databases to larger instance sizes as user traffic escalates, which typically corresponds to revenue increases and mitigates the costs of the larger instances.

In addition to these scripts that facilitate the migration of a database to a different instance size, RightScale also provides scripts to enable database tuning based on the instance size and characteristics. Our extensive experience with MySQL (both in managing our own web-based application as well as assisting our customers), has given us the opportunity to accumulate a vast array of knowledge on modifying MySQL configurations to provide increased performance based on such resources as system memory and the number of virtual cores that are available on the instance.

© 2011 RightScale, Inc. All rights reserved.

3.4.1.2

Horizontal Scaling

As I mentioned previously, we highly recommend implementing one or more slave databases in addition to the master database, regardless of the application lifecycle phase. The presence of multiple slave databases increases the overall reliability and availability of the application, as well as enables horizontal scaling of the database using a proxy mechanism for database reads, such as that provided by MySQL Proxy.

In a MySQL Proxy configuration (see **Figure 4**), no changes are necessary to the application tier. The application servers use their normal database connector, the only difference being that they are pointed at the MySQL Proxy server instead of the master MySQL server. If an application is performing a database write, the proxy server passes this request directly to the master database server. However, if a read request is being performed by the application, the MySQL Proxy server will send this request to one of the slave database servers, thus distributing the overall read load of the application.

It is important to note that replication lag to the slave databases may result in outdated data being returned in response to a read request if the read is made quickly after the data is written to the master database. For applications that rapidly write and then read the same data object, a proxy solution may not be the most effective method of database scaling.

With a MySQL Proxy implementation, although database write performance is unaffected, read performance is enhanced because all read requests are distributed among all available slaves. For applications that are read-intensive, a MySQL Proxy solution can provide a significant decrease in database load, and therefore a significant increase in application performance. Because each application is unique, we recommend that you benchmark read versus write requests throughout an application's lifecycle to see if any benefit would be gained from a database proxy solution.

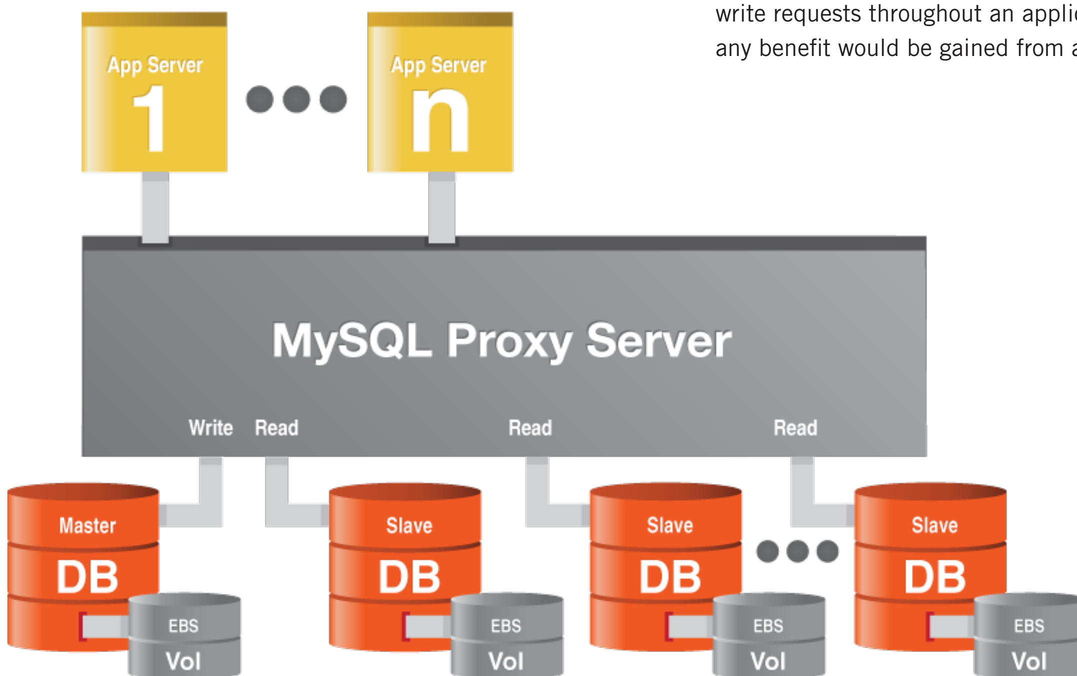


Figure 4 – Horizontal Database Scaling with MySQL Proxy

© 2011 RightScale, Inc. All rights reserved.

3.4.1.2 (cont.)

The issue with an architecture such as the one shown in **Figure 4** is that the lone MySQL Proxy server is a potential single point of failure. If this server were to suffer an outage, there would be no way for the application servers to access the database for either reads or writes. As such, the exact implementation shown in **Figure 4** is not a recommended best practice, but rather serves as a good visual representation of a proxy solution.

If you implement a MySQL Proxy solution, we recommend that you install a MySQL Proxy client on each server in the scalable application tier. The application's database connector should then be pointed at the local MySQL Proxy client, which is aware of the true location of the master database and all associated slaves. This local proxy handles the task of sending writes to the master and distributing reads to the slaves.

Another option for horizontal database scaling is database sharding, with which many RightScale customers have had success. Sharding is effectively the partitioning of your database tables into two or more distinct pieces and placing them on separate database servers.

In the simplest approach, the application servers write to a single server using a modified connector, so they do not need to be aware of which shards reside on which servers because this is handled by the sharding server (similar in concept to what is depicted in **Figure 4** for MySQL Proxy). Additionally, you can remove the sharding server and modify the application to be aware of the sharding implementation so that it can contact the appropriate database server based on the data object to be accessed.

Sharding is similar to a proxy solution in that database reads are spread across multiple servers. But due to the table partitioning, sharding also enables the database writes to be distributed, thus reducing the load on any individual database server. Depending on the application characteristics and the sharding approach you implement, you can often accomplish a fairly even distribution of database queries and dramatically improve the overall database throughput of the deployment.

Applications that rely on cross-table joins are not good candidates for sharding because these queries may involve multiple servers and can therefore be slow to process. However, most databases can be sharded if you plan for it early in the design phase. Numerous RightScale customers with high traffic volume rely heavily on database sharding to handle the loads generated by hundreds of thousands (and sometimes millions) of concurrent users. You can set up sharding yourself or work with a technology provider that has assisted many RightScale customers with sharding implementations.

In outlining best practices for horizontal database scaling, the concept of a master-master implementation bears mention as well. In a master-master design, there are (as the name implies) multiple master databases, with each master having the ability to modify any data object within the database. It is the responsibility of each master to propagate any changes to the other masters to ensure database consistency. This is in direct contrast to the master-slave implementation in which changes to the data objects can only be made by the lone master.

Although a master-master solution makes failover and failback scenarios easier to implement, it does not help in a true horizontal scaling fashion in that each master still needs to perform each and every write. It is for this reason that the master-master implementation is not a recommended best practice. With the potential for latency between masters, particularly as additional masters are added or aggressive replication systems are implemented, it is extremely difficult if not impossible to ensure consistency between masters, and thus ensure the integrity of the data. Some RightScale customers have had limited success implementing their own master-master solutions using highly modified application data processing schemes, but we do not recommend or support it.

© 2011 RightScale, Inc. All rights reserved.

3.4.1.2 (cont.)

A growing percentage of RightScale customers have begun investigating the AWS Relational Database Service (RDS) as a possible solution to their database needs. RDS is specific to the AWS infrastructure and is therefore not a portable solution. However, with the introduction of multi-AZ (availability zone) support and read replicas, RDS has become a very viable database solution. The multi-AZ implementation is twice the cost of the standard configuration but provides mechanisms for automated failover. (Note: We do not recommend a non-multi-AZ solution because up to four hours of scheduled downtime is required each week.)

Similar to the AWS ELB, RDS is an appliance and, as such, the same caveats apply: Namely, access to the underlying resources hosting the database is not possible. Thus, you cannot evaluate performance metrics (CPU utilization, free memory, slow-query logs, etc.). But if you desire a distributed, redundant database appliance with automated failover and don't require knowledge or access to the underlying details of the implementation, RDS may prove a viable solution, and should be evaluated for any new application.

© 2011 RightScale, Inc. All rights reserved.

3.4.1.3

NoSQL Solutions

Recently, many new database/datastore technologies have become more commonplace in scalable application architectures. One of the most common categories of these new technologies is NoSQL database management systems.

As the name implies, these technologies do not provide the common query interface present in SQL solutions. Instead of the common relational aspects of most SQL implementations, these technologies are essentially key/value stores such that a particular database object is mapped to a specific unique key. The object that is stored can be of virtually any format, from a simple string or integer to a serialized object or complex blob file. No relational associations exist between these objects, so NoSQL implementations are not viable database solutions for applications that require RDBMS-like functionality.

However, for applications that do not require relational tables and structured queries (such as some social gaming applications), NoSQL solutions may provide a fast, scalable solution. In NoSQL implementations, you can combine multiple nodes to provide the datastore and add nodes to enable horizontal scalability. But coordinated backup and recovery can be challenging, so you should perform a careful evaluation of the capabilities and limitations of NoSQL solutions prior to deciding on an application's database architecture.

© 2011 RightScale, Inc. All rights reserved.

4

Conclusion

As the availability and functionality provided by cloud resources advance both technically and in the breadth of their capabilities, the opportunities for scalable applications will continue to thrive. Due to the unpredictable nature of user-facing applications that are subject to the possibility of viral and flash crowd events, planning the resources necessary to support an application is a challenging and imprecise activity. With virtually limitless on-demand resources, the cloud provides an ideal infrastructure in which to implement and launch a scalable application.

However, care should be taken in the design phase to plan for the potential of an application's unprecedented success, while keeping in mind budget constraints during the initial phases of the application when resource demands are low. While no two applications are exactly alike, many share common characteristics and traits from which patterns of use can be realized, assisting you in the architectural design of these applications.

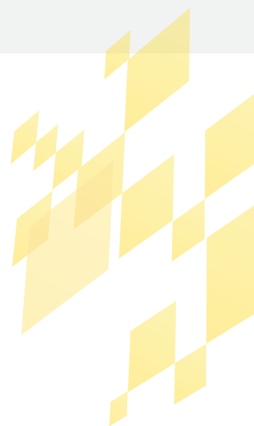
As this white paper has illustrated, there are several basic overarching concepts with regard to application architecture, and typically several different possible implementations of each of these concepts. The key to developing an application with designs for scalable long-term success is to select the components of these architectures that are relevant to the application's design and user interactions and to modify these components as required to accommodate the unique aspects of the application.

The multi-tiered architecture I present in this white paper is a compilation of the lessons learned from the numerous customer engagements in which RightScale has had the opportunity to be involved, in concert with cloud best practices around the implementations of these applications. The concepts I present here are intended to be used as a framework and reference for architecting a scalable application, and I want to make the point that no single architecture is ideal for every situation. However, the approaches presented here incorporate many of the best practices and recommended solutions that you can use as a referenceable architecture for new scalable applications destined for implementation in the cloud.

About RightScale

RightScale is the leader in cloud computing management. Founded in 2006, the company offers a fully automated cloud management platform that enables organizations to easily deploy and manage business critical applications across multiple clouds with complete control and portability. The RightScale Cloud Management Platform is delivered as "software as a service" (SaaS) and is available in a range of editions. To date, thousands of deployments and more than two million servers have been launched on the RightScale platform for leading organizations such as PBS, Harvard University, Zynga, and Sling Media. To learn more about RightScale, please go to RightScale.com.

Questions?
Worldwide Sales and Support Inquiries:
1.866.720.0208 (toll free) or
+1.805.855.0265
sales@rightscale.com



© 2011 RightScale, Inc. All rights reserved.