Chapter 14
Efficient Practices and Frameworks for Cloud-Based Application Development

Anil Kumar Muppalla, N. Pramod, and K.G. Srinivasa

Abstract As cloud computing continues to burgeon throughout the technology sphere, it becomes essential to understand the significance of this emerging technology. By its nature, it offers an organization a great deal of agility and cost savings. Cloud technologies are being applied and leveraged in different applications fueling growth in the number of Infrastructure-as-a-Service (IaaS) and Platform-as-a-Service (PaaS) vendors. The business delivery models of cloud computing have raised interests across the IT industry as the resources are offered as utilities and on demand. From a developer perspective, it is important to grasp the nuances of cloud-based application development to improve the development process. This chapter discusses best practices in relation to some of the celebrated cloud features. Furthermore, most common and well-known features of cloud frameworks are presented to aid the developer's choice. Lastly, comparative cloud-based architectural discussion on developing and deploying a Web application using industry popular frameworks is presented. Although, cloud computing as a service/development paradigm addresses several well-known issues like scalability and availability, there are several concerns with respect to security and privacy of data which has opened doors for research opportunities. Some plausible research directions are also identified.

Keywords Scalability • Cloud computing • Azure • App engine • Storage • Frameworks • Application development

A.K. Muppalla • N. Pramod • K.G. Srinivasa (✉)
High Performance Computing Laboratory, Department of Computer Science and Engineering, M S Ramaiah Institute of Technology, Bangalore, India
e-mail: anil.kumar.848@gmail.com; npramod05@gmail.com; srinivasa.kg@gmail.com

14.1 Introduction

Prior to 2007, there was a need for any large technology corporation to maintain infrastructure to fulfill the needs of the company and its clients [1]. With the emergence of cloud computing, the situation has changed. There seems to be wide acceptance in the prospect of buying infrastructure usage rather than the hardware itself with immediate cost benefits. The on-demand delivery of hardware, software, and storage as a service is termed as *cloud computing*. The union of data center hardware, software, and storage is what we will call a *cloud*. An application based on such clouds is taken as a *cloud application*. This paradigm has revolutionized the service industry with increasing support from Microsoft [2], Google [3], and IBM [4]. Three striking aspects of cloud computing are [5]:

- The impression of infinite cloud resources available on demand, thereby dismissing the need for users to plan far ahead for provisioning.
- The on-demand commitment of resources by cloud, thereby allowing companies to start small and request resources as and when the need arises.
- The pay-per-use model has encouraged ability to pay for use of computing resources on a short-term basis as needed and release them as needed.

Efforts to conceptualize cloud computing dates back to, at least, 1998 [6]. However, the adoption and promotion of cloud computing has been slow until 2007 [1]. The background of early industrial adoptions of cloud computing coincides with that of service computing [7]. Service computing [8] received worldwide support from leading companies like IBM and Microsoft [9]. The widespread adoption of cloud computing is driven by stable and mature development of technologies and computing resources. Success stories of Web services have complemented the popularity service computing, although a Web service is one such technology to fulfill the need for service orientation [7]. Many distributed computing techniques for cloud computing have been mature [10–12]. Decoupling the parts of the application environment allows for scalability on different levels; these parts are further provided to the developers as services. Based on the type of the service provided, cloud computing can be classified as Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS) [13].

Developers reap several benefits developing their application on a cloud-based programming environment provided through a PaaS provider, such as automatic scaling and load balancing, as well as integration with other services (email and authentication). Such provisions alleviate much of the overhead of developing cloud applications. Furthermore, integration of their applications with other services on-demand increases the likelihood of usage of these applications, thereby driving the need to develop cloud-based applications. This in turn makes the cloud application development a less complicated task, accelerates the deployment time, and minimizes the logic faults in the application, for example, deployment of a distributed computing environment such as Hadoop [14, 26] on the cloud which provides its application developers with a programming environment, that is, MapReduce

Table 14.1 Comparison between traditional and cloud-based application

Traditional applications	Cloud-based applications
Each application is deployed and maintained as a bundle in a common environment	With diverse environment capabilities of the cloud, the application is deployed and maintained as modules, scattered across environments
Run-time infrastructure is structured and controlled, giving rise to maintenance overhead	Run-time infrastructure is unstructured and managed by cloud fabric, with computing capabilities changing
Business functionality is realized by using “controller” components that calls methods (functions) of business components	Service orchestration is used to realize business functionality—invoke one or many business services
Support for multi-tenancy is typically not required	Multi-tenancy support is assumed
User base is assumed at design time, and scalability is addressed at run-time by procuring necessary hardware	User base need not be known, potential to scale up and down rapidly
Enhancements and upgrades require downtime	No downtime required for enhancements and upgrades
Components interact with non-SOA contracts like RMI and CORBA	Standard SOA service-based interaction between components is assumed like SOAP and REST
Deployment requires traditional tools (application server admin console, ANT, etc.)	Along with traditional tools, requires knowledge and utilization of vendor-specific cloud APIs
Application is tested in controlled environment (Unit/integration/system)	Application (integration) is tested on the cloud to ensure seamless orchestration between services on one or many clouds
Security is enforced by application architecture (LDAP lookup based authentication/authorization)	Security is built into the service contracts (WS-Security, SAML, etc.)

framework for the cloud. As such, cloud software environments facilitate the process of cloud application development.

Cloud computing brings this whole new way of thinking about architecture and design, since we don’t control the infrastructure directly hence one step less in the design process. The application is supported to scale horizontally, be very cost effective in operation as you can scale up and scale down and obtain granular control over CPU expense. As several platforms such as Force.com are rich and provide the boilerplate code, developing applications on it becomes a much higher-level activity. The gap between domain experts who conceptualize the product and developers who code it significantly narrows down. The adoption of cloud computing has improved the development process of several applications and services.

The differences between cloud-based application and traditional application are presented in Table 14.1. There is no significant change in the development process of a cloud application; since the division of the application development environment into infrastructure, platform, and software has significantly helped in overcoming some common challenges of traditional software development, it has led to accelerated development and deployment, ensuring shorter release cycles. The cloud application development enforces an agile form of development. Some advantages are:

- Short release cycles means processes used for developing these applications are agile/scrum based.
- Heavy stress on acceptance as well as unit tests.
- Traditional task management practices and timesheet processes are not applicable.
- No formal workflow processes for reviews.

14.2 Design Patterns for Key Issues of Cloud Application Development

14.2.1 Scalability

This is defined as the ability of the system to handle growing amount of work in a reliable manner [15]. Scalability in cloud perspective can be addressed by considering the following:

14.2.1.1 Load Sharing

It is the logical spreading of requests across similar components for handling those requests, from a cloud development point of view, and distribution of requests, which are mainly HTTP but can be any application protocol, across all the instances using an efficient configured load-balancing algorithm. This is a scaling-out approach. Several load-balancing facilities are provided across development platforms; the task of the developer would be to tie the application to these APIs.

14.2.1.2 Partitioning

Intelligent load distribution across many components by routing an individual request to a data-specific component, efficiency, and performance is dramatically increased in an application's delivery architecture while enabling this facility. Instead of having identical instances, each instance or pool of instances, as shown in Fig. 14.1, is marked as the *owner*. This enables the developers to configure the development environment to handle type-specific request. The concept of *application switching* and *load balancing* achieve individual importance as the former is used to route a particular request which can be then load balanced across a pool of resources. It's a subtle distinction but an important one when architecting not only efficient and fast but resilient and reliable delivery networks.

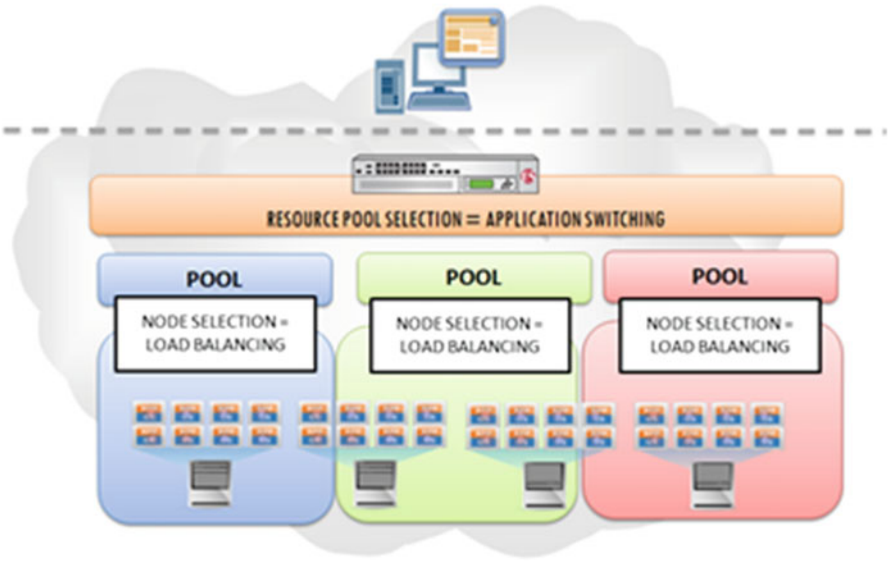
RESOURCE POOL SELECTION = APPLICATION SWITCHING
POOL
NODE SELECTION = LOAD BALANCING
POOL
NODE SELECTION = LOAD BALANCING
POOL
NODE SELECTION = LOAD BALANCING

Fig. 14.1 Grouping instances into task-specific pools [16]

14.2.1.3 Vertical Partitioning

It is a partitioning using different processing units while routing application requests that we separate by function that is associated with a URI. Content wise, partitioning is the most common implementation strategy. Consider an example of creating resource pools based on the Content-Type HTTP header: content in pool *content servers* and images in pool *image servers*. This provides for greater optimization of the Web/application based on the usage pattern and the content type. In a distributed environment, architects leverage say cloud-based storage for static content while maintaining dynamic content (and its associated data stores) on premise. This hybrid strategy is regarded to have successful acceptance across the cloud community.

14.2.1.4 Horizontal Partitioning

Through partitioning, persistence-based load balancing is accomplished, as well as the handling of object caching. This also describes the way in which you might direct requests received from specific users to designated instances that are specifically designed to handle their unique needs or requirements, for example, separation of *privilege* users from *free* users based on some partitioning key, which is cookie information.

14.2.1.5 Relaxing Data Constraints

Techniques and trade-offs with regard to the immediacy of processing/storing/access to data fall in this strategy. This requires intelligent handling of data storage and access based on varying properties like usage and prioritization of the content. If one relaxes the constraints around access times for certain types of data, it is possible to achieve a higher-efficiency use of storage by subjugating some content to secondary and tertiary storage tiers which may not have the same performance attributes as your primary storage tier.

Architecting a solution that separates data reads from writes implies eventual consistency, as data updated/written to one database must necessarily be replicated to the databases from which reads are, well, read, but that's part of relaxing a data constraint.

14.2.1.6 Parallelization

This refers to working on the same task in parallel on multiple processing units employing tools and methods like MapReduce and SPDY. If the actual task can be performed by multiple processing units, then an application delivery controller could certainly be configured to recognize that a specific URL should be essentially sent to some other proxy/solution that performs the actual distribution. We can observe that the processing model here deviates sharply from the popular *request-reply* paradigm.

14.2.1.7 Going Stateless

Application state maintenance can often hinder any scalability efforts, which normally involves persistence, and persistence means storing your data in some central location, and central data store is difficult to scale. Adopting RESTful nature (without being limited to HTTP) is a viable choice.

14.2.2 Elasticity

Dynamic resource utilization is a central concept in cloud computing. Application design must allow resources to be reserved and freed as needed. The aspects that drive the need to automate elasticity are as follows: (1) applications have to monitor themselves or have to be monitored externally, (2) application resources have to be provisioned based on this information, and (3) applications have to cope with addition and removal of resources. In order to fully benefit from the dynamicity of an elastic infrastructure, the management process to scale out an application has to be automated [17]. This way, the number of used resources can be aligned to changing workload quickly. If pay-per-use pricing models are available, the resource number directly affects the running cost of the application. Manual resource scaling would not respect this.

Requests received by an application are a good measure of workload and therefore shall be used as a basis for scaling decisions. An elastic load balancer automatically determines the amount of required resources based on numbers of requests and provisions the needed resources accordingly using the elastic infrastructure's API. Number of requests in unit time is observed from the components, and required number of resources (this is crucial design element) is computed by the load balancer and provisioned on the elastic infrastructure using its API. It significantly affects the effectiveness of the scaling decisions. It should be carefully selected during the design of the application using capacity planning techniques. Also, such behavior needs to be real time.

If the application can handle asynchronous requests, another layer of optimization can be implemented since there is a possibility of fluctuation in resource costs or cloud elasticity. The tasks can be delayed based on the availability of the resources. Some non-business-critical or time-critical workload, such as report generation, can be moved to times when resources of the private cloud are less utilized. An *elastic queue* is used to distribute requests among application components. Based on the number and type of messages it contains, the elastic queue determines the number of computing nodes to be provisioned. The elastic queue can contain different message types that are handled by different components. To speed this process up, individual images for application components are stored in the image database of the elastic infrastructure. Additionally, the elastic queue can respect environmental information, such as the overall infrastructure or resource price. This is used to delay less critical messages by reducing the number of handling compute nodes and to prioritize the business-critical functionality if the overall infrastructure utilization is high.

14.2.3 Availability

The use of commodity hardware to build the cloud has an advantage to reduce costs but also reduces the availability of resources. Therefore, cloud applications have to be designed to cope with failing resources to guarantee the required availability. Sometimes, (high) availability is only expressed regarding the possibility to start new compute nodes. To guarantee high availability under such conditions, the application architecture needs to be adjusted to enable redundancy and fault-tolerant structures. The application architecture is altered to include redundant compute nodes performing the same functionality. High available communication between these nodes is assured, for example, by a messaging system. Additionally, compute nodes are monitored and replaced in case of failure.

In a setting where high available compute nodes are used, the decoupling of components can also increase the performance and enable elasticity. As in every setup where messaging is used, the compute nodes need to consider the delivery assurances made by the messaging systems. Business-critical components of an application should be available at all times even during update. During an update, the elasticity of cloud aids in provision of additional compute nodes that contain the new application or middleware versions additionally to the old versions, consequently the shutdown of old compute nodes. One such method is providing images for compute nodes with the new software version that is created and tested. Hence, a graceful transition from the old to new application versions is executed. If different versions must not be handling requests at the same time, the transition is imminent. This is handled by instantiating both application versions independently. The switch can then be made by reconfiguring the access component, such as a load balancer. However, in some cases this can result in a minimal downtime during the transition [17].

14.2.4 Multi-tenancy

Any party that uses an application is termed a *tenant*. Sometimes a tenant can be a single user of an entire organization. Many of the cloud properties, such as elasticity and pay-per-use pricing models, can only be achieved by leveraging economies of scale. Cloud providers therefore have to target large markets and share resources between customers to utilize resources effectively. Hardware virtualization has been the first to foray into resource sharing through *Infrastructure-as-a-Service* delivery model. There is need for additional architectural modifications to support sharing of higher-level application components. When application is provided to multiple customers (multi-tenacity), deployment of componentized applications can be optimized by sharing individual component instances whenever possible. This is especially feasible for application components that are configured equally for all tenants, for example, currency converters. If tenants can share common resources, then underlying resources can be utilized in more efficient ways. This requires the configuring for multi-tenacity. The tenant's individual application instances access the same application component (pool). Therefore, the run-time cost per tenant can be reduced, because the utilization of the underlying infrastructure is increased and the shared component can be scaled for all tenants. If the configuration is equivalent for all tenants, a *single instance* can be used. Sometimes, tenants are not allowed to share critical components with other users. In this case, a *multiple instance component* must be used.

Additional use case wherein an application is instantiated to support multi-tenacity but some of its components cannot be shared may be due to laws prohibiting the same. So, tenants may require integration of individually developed application components into the provided application. Deploy individual component implementations and configurations for each tenant. This arrangement allows tenants to adjust components very freely. Portions of an application, on which tenants have a versatile behavior, can be realized in such a fashion. However, the application of this pattern hinders resource sharing between tenants.

14.2.5 High Performance

A load-balancing algorithm coupled with the MapReduce programming paradigm serves the purpose of processing large volumes of data. MapReduce is a parallel programming model that is supported by some capacity-on-demand type of clouds such as Google's BigTable, Hadoop, and Sector [18]. Load balancing is helpful in spreading the load equally across the free nodes when a node is loaded above its threshold level. Though load balancing is not so significant in execution of a MapReduce algorithm, it becomes essential when handling large files for processing and when availability of hardware resources is critical. Hadoop MapReduce has wide industry acceptance also being the top programming model implemented.

An efficient load-balancing technique can sometimes make all the difference in obtaining maximum throughput. The arrangement is considered balanced if for each data node, the ratio of used space at the node to the total capacity of node (known as the *utilization of the node*) differs from the ratio of used space at the cluster to the total capacity of the cluster (*utilization of the cluster*) by no more than the threshold value [17]. In view of hyper-utilization the module moves blocks from the data nodes that are being utilized a lot to the poorly used ones in an iterative fashion. In this implementation, nodes are classified as *high*, *average*, and *low* depending upon the utilization rating of each node. In a cloud environment, the MapReduce structure increases the efficiency of throughput for large data sets. In contrast, you wouldn't necessarily see such an increase in throughput in a non-cloud system. Therefore, consider a combination of MapReduce-style parallel processing and load balancing when planning to process a large amount of data on your cloud system.

14.2.6 Handling Failure

Unlike the traditional applications which are entirely dependent on the availability of the underlying infrastructure, cloud applications can be designed to withstand even big infrastructure outages. With the goal that each application has minimal or no common points of failure, the components must be deployed across redundant cloud components. These components must make no assumptions about the underlying infrastructure; that is, it must be able to adapt to changes in the infrastructure without downtime.

Designing for failure also comes with fair share or challenges such as large data processing which requires frequent movement of large volumes of data causing inertia. By building simple services composed of a single host, rather than multiple dependent hosts, one can create replicated service instances that can survive host failures. For example, if we had an application that consisted of business logic component 1, 2, 3, each of which had to be live on a separate host, we could compose service group (1, 2, 3), (1, 2, 3)… or we could create component pools (1, 1, …), (2, 2, …), (3, 4, …). While the composition (1, 2, 3), a single machine failure would result in the loss of a whole system group. By decomposing resources into independent pools, a single host failure only results in the loss of a single host's worth of functionality.

Another practice is to ensure short response time ensured by noting if the request returns a transient error or doesn't return within a small time, a retry is triggered to another instance of the service. If you don't fail fast and retry, distributed systems, especially those that are process or thread-based, can lock up as resources are consumed waiting on slow or dead services.

Thus, separating business logic into small stateless services that can be organized in simple homogeneous pools is much more efficient. The pool of stateless recording services allows upstream services to retry failed requests on other instances of the recording service. In addition, the size of the recording server pool can easily be scaled up and down in real time based on load.

14.3 Analysis of Storage as a New Form of Service

As technology continues to mature, several previously coupled components have broken out to exist independently. One such component is storage, still part of the infrastructure in principle, which has open doors for targeting specific business areas. To understand the application of storage as a service on its own, several delivery metrics need to be discussed along with established best practices [27], with support of the general architecture in Fig. 14.2.

14.3.1 Access

One problem with Web service APIs is that they require integration with an application to take advantage of the cloud storage. Most providers implement multiple access methods, but Web service APIs are common. Many of the APIs are implemented based on REST principles, which imply an object-based scheme developed on top of HTTP (using HTTP as a transport). REST APIs are stateless and therefore simple and efficient to provide. Therefore, common access methods are also used with cloud storage to provide immediate integration. For example, file-based protocols such as NFS/Common Internet File System (CIFS) or FTP (File Transfer Protocol) are used, as are block-based protocols such as iSCSI (Internet Small Computer System Interface).

14.3.2 Performance

Performance issues of storage systems range from small transactional accuracy to large data movement, but the ability to move data between a user and a remote cloud storage provider represents the largest challenge from a cloud storage perspective.

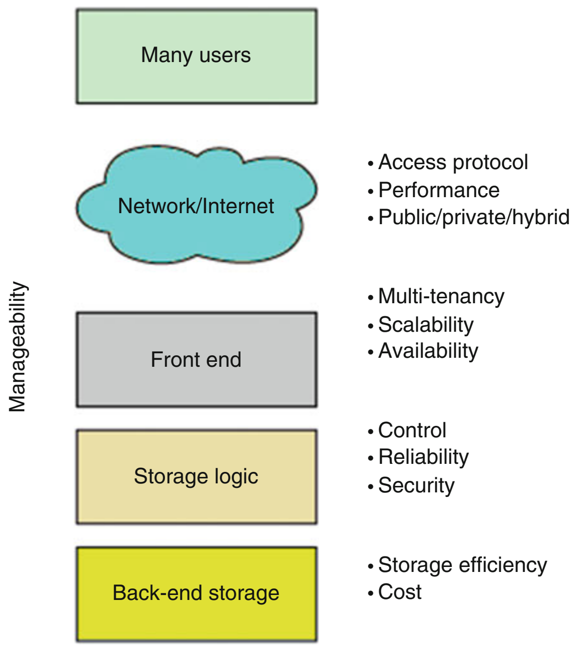

Fig. 14.2 General architecture of storage service [17]

The problem is TCP, as it controls the flow of data based on packet acknowledgments from the peer endpoint. Packet loss and late arrival enable congestion control as a useful feature but also limits performance as these are more network-intensive tasks. TCP is ideal for moving small amounts of data through the global Internet but is less suitable for larger data movement, with increasing RTT (round-trip time). This problem is solved by removing TCP from the equation. A new protocol called the *Fast and Secure Protocol* (FASP) was developed to accelerate bulk data movement in the face of large RTT and severe packet loss. The key is the use of the UDP, which is the partner transport protocol to TCP. UDP permits the host to manage congestion, pushing this aspect into the application layer protocol of FASP, as shown in Fig. 14.3.

14.3.3 Availability

Once a cloud storage provider has a user's data, he/she must be able to provide that data back to the user upon request. Given the network outages, user errors, and other circumstances, reliability and availability can prove to be a major hurdle. There are some interesting and novel schemes to address availability, such as information dispersal (Information Dispersal Algorithm (IDA)), to enable greater availability of data in the face of physical failures and network outages. IDA is an algorithm that

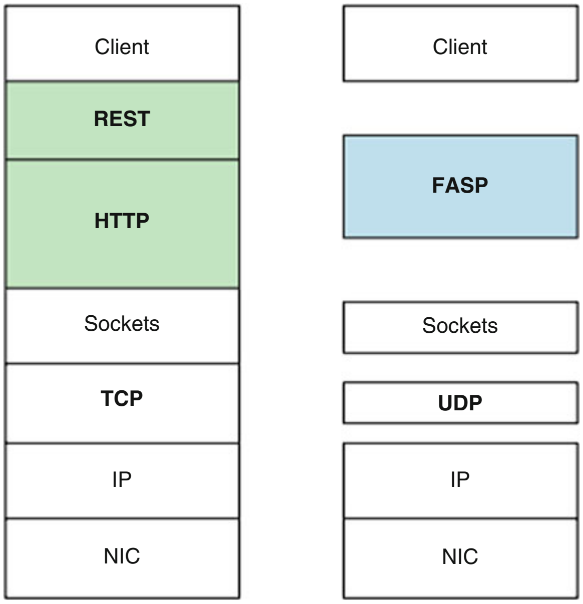

Fig. 14.3 Communication stack

allows data to be sliced with Reed-Solomon codes for purposes of data reconstruction in the face of missing data. Furthermore, IDA allows you to configure the number of data slices, such that a given data object could be carved into four slices with one tolerated failure or 20 slices with eight tolerated failures. Similar to RAID, IDA permits the reconstruction of data from a subset of the original data, with some amount of overhead for error codes (dependent on the number of tolerated failures). The downside of IDA is that it is processing intensive without hardware acceleration. Replication is another useful technique and is implemented by a variety of cloud storage providers. Although replication introduces a large amount of overhead (100 %), contrast to very low overhead by IDA, it is simple and efficient to provide.

14.3.4 Control

A customer's ability to control and manage how his or her data is stored has always motivated several storage providers. Although replication is a common method to ensure redundancy and hence availability, it also requires more than idea storage space. Reduced Redundancy Storage (RRS) is one such method that ensures to provide users with a means of minimizing overall storage costs. Data is replicated within the vendor's infrastructure, but with RRS, the data is replicated fewer times with the possibility for data loss. This is ideal for data that can be recreated or that has copies that exist elsewhere.

14.3.5 Efficiency

Storage efficiency is an important characteristic of cloud storage infrastructures, particularly with respect overall cost. This characteristic speaks more to the efficient use of the available resources over their cost. To make a storage system more efficient, more data must be stored. A common solution is data reduction, whereby the source data is reduced to require less physical space. Two means to achieve this include *compression*—the reduction of data through encoding the data using a different representation—and *de-duplication*, the removal of any identical copies of data that may exist. Although both methods are useful, compression involves processing (re-encoding the data into and out of the infrastructure), where de-duplication involves calculating signatures of data to search for duplicates.

14.4 Frameworks

Developers can use the cloud to deploy and run applications and to store data. On-premises applications can still use cloud-based resources. For example, an application located on an on-premises server, a rich client that runs on a desktop computer, or one that runs on a mobile device can use storage that is located on the cloud. Cloud application development is aided significantly with the provision of frameworks and development environments which the developers can leverage to produce applications guided by useful abstractions. These frameworks have proven to reduce the development time, therefore receiving wide acceptance. The period from 2007 to 2011 has witnessed exponential growth in adoption of cloud frameworks with Amazon kicking off this trend and recently several others perfecting it. This section provides important features of three such frameworks from industry leaders like Amazon, Google, and Microsoft.

14.4.1 Windows Azure

The Windows Azure platform by Microsoft Corporation provides hardware abstraction through virtualization. Every application that is deployed to Windows Azure runs on one or more virtual machines (VMs) [19]. The applications behave as though they were on a dedicated computer, although they might share physical resources such as disk space, network I/O, or CPU cores with other VMs on the same physical host; this is the abstraction that is possible with decoupling infrastructure from the application. A key benefit of an abstraction layer above the physical hardware is portability and scalability. Virtualization of a service allows it to be moved to any number of physical hosts in the data center. By combining virtualization technologies, commodity hardware, multi-tenancy, and aggregation of demand,

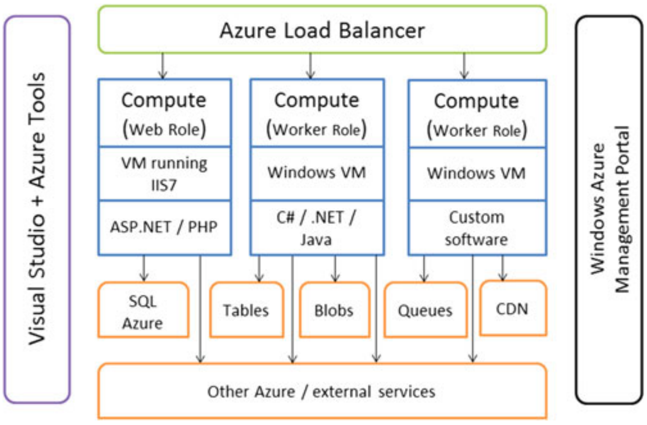

Fig. 14.4 Azure architecture [20]

Azure has become one of the most coveted platforms. These generate higher data center utilization (i.e., more useful work-per-dollar hardware cost) and, subsequently, savings that are passed along to you. Figure 14.4 presents the high-level architecture of Azure, which encapsulates the above-discussed features.

14.4.1.1 Salient Features of Azure

Here are some salient features of Windows Azure:

- Supports all major .NET technologies and provides wide language support across Java, PHP, and Python [24, 25]
- Windows Azure Compute:
  - Computing instances run Windows OS and applications (CPU + RAM + HDD)
  - Web role: Internet information services machine for hosting Web applications and WCF services
  - Worker role: long-running computations
- Azure data storage services:
  - Azure table storage: distributed highly scalable cloud database (stress entries with properties)
  - Azure queue storage: message queue service
  - Azure blobs/drives: blob/file storage, NTFS volumes
- SQL Azure: SQL server in the cloud with highly available and scalable relational database
- Azure Business Analytics: create reports with tables, charts, maps, etc.
- Azure Caching: distributed, in-memory, application cache

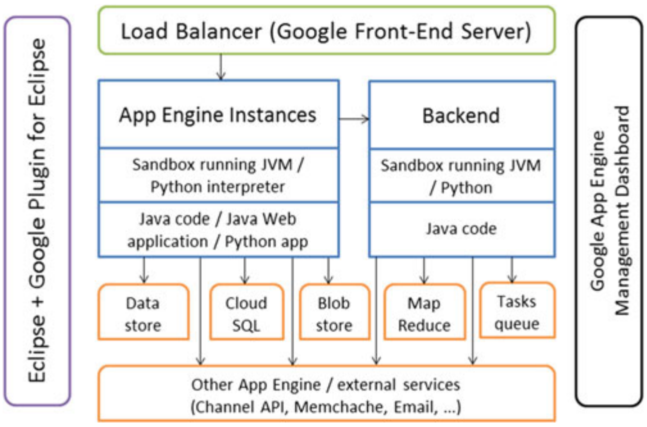
Load Balancer (Google Front-End Server)
App Engine Instances
Sandbox running JVM / Python interpreter
Java code / Java Web application / Python app
Backend
Sandbox running JVM / Python
Java code
Data store
Cloud SQL
Blob store
Map Reduce
Tasks queue
Other App Engine / external services (Channel API, Memchache, Email, ...)
Eclipse + Google Plugin for Eclipse
Google App Engine Management Dashboard

Fig. 14.5 Google App Engine architecture [20]

14.4.2 Google App Engine

Google App Engine is a Platform-as-a-Service (PaaS) cloud-computing delivery model for developing and hosting Web applications in Google-managed data centers. Applications are sandboxed and run across multiple servers [21]. App Engine offers automatic scaling for Web applications—as the number of requests increases for an application, App Engine automatically allocates more resources for the Web application to handle the additional demand [22]. Figure 14.5 represents the high-level architecture of Google App Engine outlining the structure to aid application development.

14.4.2.1 Salient Features of App Engine

- Leading Java and Python public cloud service
- App Engine instances:
  - Hosting the applications
  - Fully managed sandboxes (not VMs)
  - Provide CPU + RAM + storage + language run-time
- App Engine Backend:
  - Higher computing resources
  - Used for background processing
- App Engine data stores:
  - NoSQL schema less object database
  - Support transacts and a query engine (GQL)

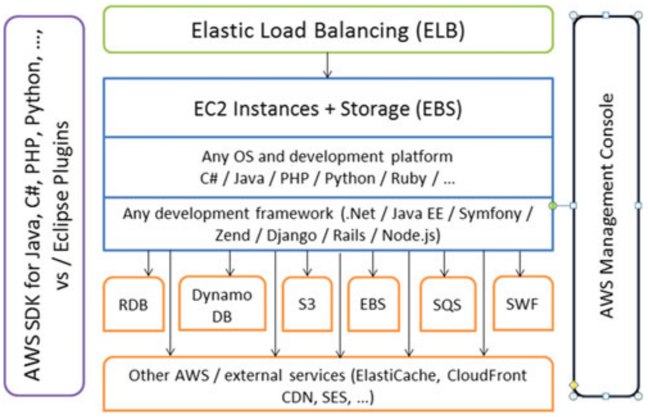
AWS SDK for Java, C#, PHP, Python, …, vs / Eclipse Plugins
Elastic Load Balancing (ELB)
EC2 Instances + Storage (EBS)
Any OS and development platform
C# / Java / PHP / Python / Ruby / …
Any development framework (.Net / Java EE / Symfony / Zend / Django / Rails / Node.js)
RDB
Dynamo DB
S3
EBS
SQS
SWF
Other AWS / external services (ElastiCache, CloudFront CDN, SES, …)
AWS Management Console

Fig. 14.6 AWS architecture [20]

- Cloud SQL: managed MySQL in App Engine
- Cloud Storage: store files as blobs and files with REST API
- MapReduce API: highly scalable parallel computing API for heavy computing tasks (based on Hadoop)
- Channel API: push notification for JavaScript applications
- Task Queues: execution of background services
- Memchache: distributed in-memory data cache

14.4.3 Amazon Web Services (AWS)

This is a collection of remote computing services (also called Web services) which constitute the cloud-computing platform provided by Amazon. Figure 14.6 represents the aggregation of wide range of features that support cloud application development on Amazon framework.

14.4.3.1 Salient Features of AWS

- Amazon Elastic Compute Cloud (Amazon EC2):
  - Virtual machines on-demand running Windows/Linux/other OS
  - Geographically distributed
  - Elastic IP addresses: a user can programmatically map an Elastic IP address to any virtual machine instance without a network administrator's help and without having to wait for DNS to propagate the new binding
- Amazon Elastic Block Store (Amazon EBS):
  - Virtual HDD volumes
  - Used with EC2 to keep the OS file system

- Amazon Simple Storage Service (Amazon S3):
  - Host binary data (images, videos, files, etc.)
  - REST API for access via Web
- Amazon DynamoDB/SimpleDB:
  - Managed NoSql cloud database
  - Highly scalable and fault tolerant
- Amazon Relational Database Service (RDS):
  - Managed MySQL and Oracle databases
  - Scalability, automated backup, replication
- Other services:
  - SQS: message queue
  - CloudFront: content delivery network
  - ElastiCache: caching
  - Route 53: Cloud DNS
  - SES: email

14.5 Comparison of AWS and Windows Azure: Applications Development

While deploying an initial Web application on the cloud, care is taken to leverage the niche technologies provided by the environment. This section performs a comparative analysis of the above-mentioned features in building a Web application on Amazon Web Services against Windows Azure.

14.5.1 Local Application Development Setup

Apache is an application server with development in PHP and storage in MySQL database. Figure 14.7 depicts the primary setup.

14.5.2 Migrating to the Cloud

AWS: In AWS, this means an Amazon EC2 Instance, an Elastic IP, and backups to the Amazon S3 storage service.

Windows Azure: In Windows Azure, the counterpart to EC2 is Windows Azure Compute. Specify a role (hosting container) and number of VM instances.

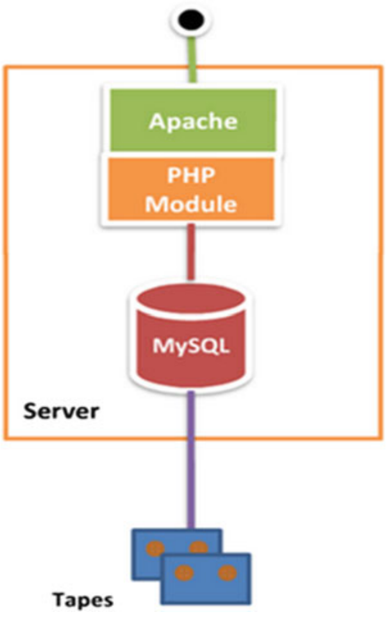

Fig. 14.7 Local application setup [23]

Choose a worker role (the right container for running Apache) and one VM instance. Upload metadata and an application package, from which Windows Azure Compute, Windows Server VM instance is created. An input endpoint is defined which provides accessibility to the Web site. Backups are made to the Windows Azure Storage service in the form of blobs or data tables.

14.5.3 Design for Failure

Keep application logs and static data outside of the VM server by using a cloud storage service. Make use of database snapshots, which can be mapped to look like drive volumes as in Figs. 14.8 and 14.9.

AWS: The logs and static data are kept in the Amazon S3 storage service. Root and data snapshot drive volumes are made available to the VM server using the Amazon Elastic Block Service (EBS).

Windows Azure: Logs and static data are written to the Windows Azure Storage service in the form of blobs or tables. For snapshots, a blob can be mapped as a drive volume using the Windows Azure Drive service. As for the root volume of the VM, this is created from the Windows Azure Compute deployment just as in the previous configuration.

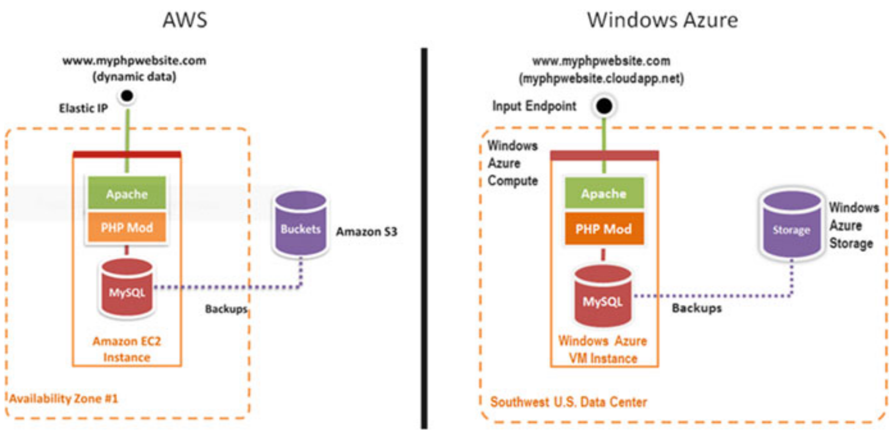

Fig. 14.8 Application deployment in AWS and Azure [23]

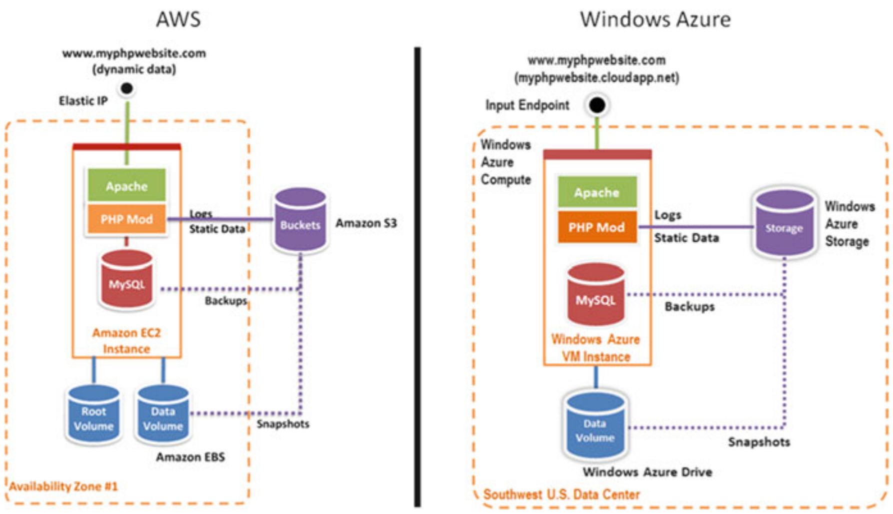

Fig. 14.9 Updated figure—design for failure [23]

14.5.4 Content Caching

Take advantage of edge caching of static content. Use content distribution network to serve up content such as images and video based on user location as in Fig. 14.10.

AWS: Amazon CloudFront is the content distribution network.

Windows Azure: The Windows Azure Content Delivery Network (CDN) can serve up blob content using a network of 24+ edge servers around the world.

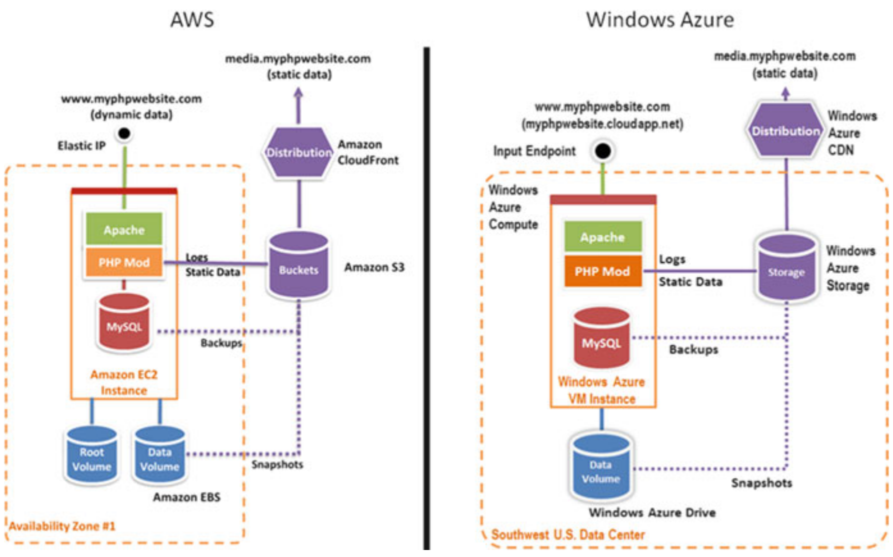

Fig. 14.10 Updated figure—caching static content [23]

14.5.5 Scaling Database

In preparing to scale, the setup must move beyond a self-hosted database on a single VM server instance. By using a database service outside of the compute VM, use multiple compute VMs without regard for data loss as in Fig. 14.11.

AWS: The Amazon Relational Database Service (RDS) provides a managed database. Andy can continue to use MySQL.

Windows Azure: Switch over to SQL Azure, Microsoft's managed database service. Data is automatically replicated such that there are three copies of the database.

14.5.6 Scaling Compute

With a scalable data, scale the compute tier, which is accomplished by running multiple instances as in Fig. 14.12.

AWS: Multiple instances of EC2 through the use of an Auto-Scaling Group. Load-balancing Web traffic to the instances by adding an Elastic Load Balancer (ELB).

Windows Azure: The input endpoint comes with a load balancer. The worker role is a scale group—its instances can be expanded or reduced, interactively or programmatically. The only change that needs to be made is to increase the worker roles instance count; a change can be made in the Windows Azure management portal.

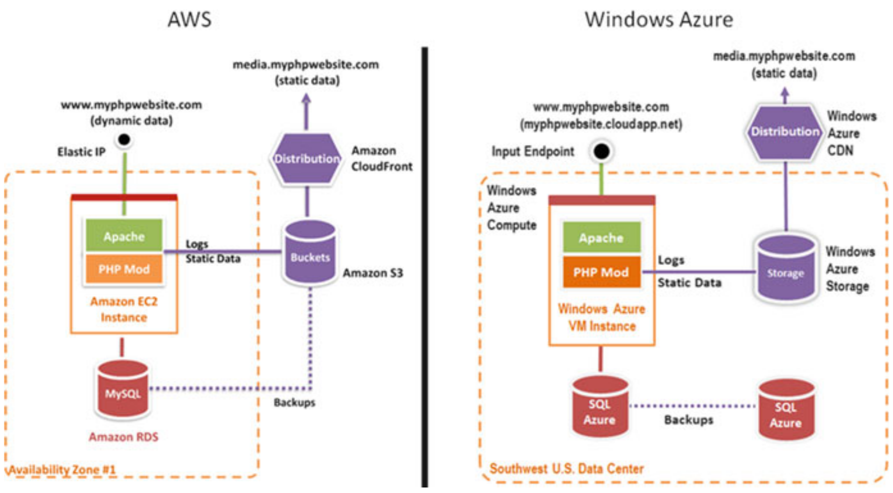

Fig. 14.11 Updated figure—database service [23]

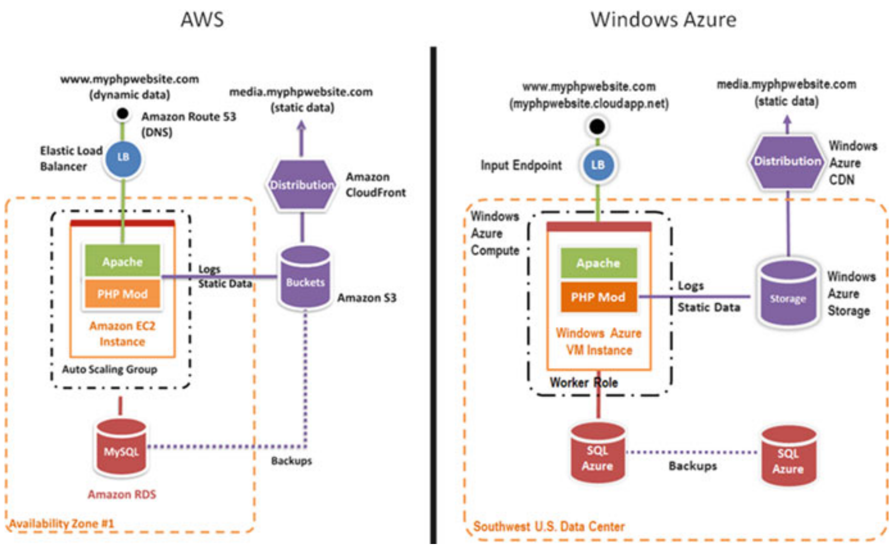

Fig. 14.12 Updated deployment—compute elasticity [23]

14.5.7 Failover

To keep the service up and running in the face of failures, one must take advantage of failover infrastructure as in Fig. 14.13.

AWS: The primary Amazon RDS database domain has a standby slave domain. The solution can survive the failure of either domain.

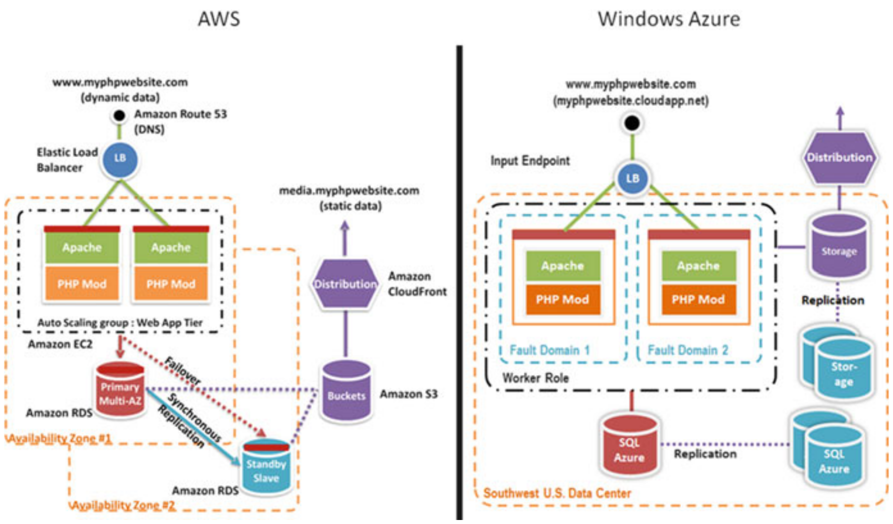

Fig. 14.13 Updated deployment—fault tolerant [23]

Windows Azure: The Windows Azure infrastructure has been providing *fault domains* all along. Storage, database, and compute are spread across the data center to prevent any single failure from taking out all of an application's resources. At the storage and database level, replication, failover, and synchronization are automatic. Since the compute was only one instance, this could be a possible hurdle, which can be addressed by running at least two instances in every role.

14.6 Future Research

The future of cloud computing continues to show promise and gain popularity. One should be able to *plug in* an application to the cloud in order to receive the power it needs to run, just like a utility. As an architect, you will manage abstract compute, storage, and network resources instead of physical servers. Scalability, security, high availability, fault tolerance, testability, and elasticity will be configurable properties of the application architecture and will be an automated and intrinsic part of the platform on which they are built.

However, we are not there yet. Today, you can build applications in the cloud with some of these qualities by implementing the best practices highlighted in this chapter. Best practices in cloud-computing architectures will continue to evolve, and as researchers, we should focus not only on enhancing the cloud but also on building tools, technologies, and processes that will make it easier for developers and architects to plug in applications to the cloud easily.

The challenge of transitioning from your local development environment seems to bother every developer; it is difficult to transition from doing stuff locally and trying it out to working in the cloud. The maturity of IDEs that can handle cloud environment is still a work in progress as well. The more seamless the transition from the local test environments to cloud-based environments, the more productive the development cycles will be. Another challenge is data security; as the application will be hosted on third-party infrastructure, the safety of the data is always at risk. There is a greater need to address this necessity both at the application level and infrastructure level.

14.7 Conclusion

Cloud-based application development process has its share of advantages and disadvantages, but many of the inherent issues are alleviated by following the basic design patterns and frameworks described in this chapter.

We can enumerate the reasons to choose either of the frameworks mentioned, clearly because the type of application that needs to be developed requires that right kind of environment. Reasons to use GAE (Google App Engine) are:

- You don't need to pay until you see a visible need to scale.
- Google services like Gmail and Calendar plug in are very easy.
- Good choice if Python or Java is used as a language.
- Locally tested app runs as is on GAE.
- Allows running multiple versions of on the same data store.

Reasons to use Azure are:

- Better suited for SOA (Service-Oriented Architecture)-based applications
- Application staging feature helps during deployment
- Two storage solutions—SQL Azure (relational) and Azure Storage (non-relational)
- Best suitable for .NET-based applications

Reasons to use Amazon Web Services are:

- Have footprint across several Linux distributions and also Windows support, while Azure allows Windows only
- Have support for myriad language platforms like C#, PHP, ASP.NET, Python, and Ruby
- Provide off-the-shelf load balancing, varying storage sizes to instances, and install custom software

While making the choice of a platform, several reasons, as listed above, need to be considered to aid in the efficient cloud application development.

References

1. Cloud computing: http://en.wikipedia.org/wiki/Cloud_computing (2008)
2. Nytimes: Software via the Internet: Microsoft in 'cloud' computing Microsoft Corporation. http://www.nytimes.com/2007/09/03/technology/03cloud.html (2007)
3. Baker, S.: Google and the wisdom of clouds. http://www.businessweek.com/magazine/content/07_52/b4064048925836.htm (2007)
4. Big blue goes for the big win: http://www.businessweek.com/magazine/content/08_10/b4074063309405.htm (2009)
5. Armbrust, M., Fox, A., Griffith, R.: A view of cloud computing. Commun. ACM **53**(4), 50–58 (2010)
6. Chellappa, R.: Cloud computing: emerging paradigm for computing. In: INFORMS 1997, Dallas, TX (1997)
7. Benatallah, B., Dijkman, R.M., Dumas, M., Maamer, Z.: Service-composition: concepts, techniques, tools and trends. In: Z. Stojanovic, A. Dahanayake (eds) Service-Oriented Software System Engineering: Challenges and Practices, pp. 48–66. Idea Group, Hershey (2005)
8. Stevens, M.: Service-oriented architecture introduction. http://www.developer.com/services/article.php/1010451 (2009)
9. Service orientation and its role in your connected systems strategy. Microsoft Corporation. http://msdn.microsoft.com/en-us/library/ms954826.aspx (2004)
10. Buyya, R.: Economic-based distributed resource management and scheduling for grid computing. Ph.D. thesis, Chapter 2. Monash University, Melbourne (2002)
11. Dell cloud computing solutions: http://www.dell.com/cloudcomputing (2008)
12. Foster, I., Kesselman, C., Tuecke, S.: The anatomy of the grid: enabling scalable virtual organization. Int. J. High Perform. Comput. Appl. **15**(3), 200–222 (2001)
13. Buyya, R., Ranjan, R., Calheiros, R.N.: Modeling and simulation of scalable cloud computing environments and the Cloudsim toolkit: challenges and opportunities in high performance computing\& simulation. In: HPCS'09. International Conference (2009)
14. Hadoop: http://hadoop.apache.org/ (2007)
15. Bondi, A.B.: Characteristics of scalability and their impact on performance. In: Proceedings of the 2nd International Workshop on Software and Performance, Ottawa, ON, Canada, ISBN 1-58113-195-X, pp. 195–203 (2000)
16. Lu, W., Jackson, J., Barga, R.: Azureblast: a case study of developing science applications on the cloud. In: Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing. ACM, New York (2010)
17. Fehling, C., Leymann, F., Mietzner, R., Schupeck, W.: A Collection of Patterns for Cloud Types, Cloud Service Models, and Cloud-Based Application Architectures in Institute Architecture of Application Systems (IAAS) Report, Daimler A G (2011)
18. Load balancing and MapReduce: http://www.ibm.com/developerworks/cloud/library/cl-mapreduce (2011)
19. Chappell, D., Windows Azure and ISVs, Technical report, Microsoft: http://www.microsoft.com/windowsazure/whitepapers (2009)
20. Svetin Nakov: Cloud for Developers Azure vs Google App Engine vs Amazon vs Appharbor, slideshare.com (2012)
21. Google: Python Runtime Environment, Google App Engine, Google Code, code.google.com (2011)
22. Sanderson, D.: Programming Google App Engine: Build and Run Scalable Web Apps on Google's Infrastructure. O'Reilly Media, Sebastopol (2009). ISBN 978-0-596-52272-8
23. David: Comparative study of AWS and Azure. http://davidpallmann.blogspot.in/2011_03_01_archive.html (2011). Accessed 23 Aug 2012
24. Microsoft Documentation: http://msdn.microsoft.com
25. User Blogs, Microsoft Documentation: http://blogs.msdn.com

26. Olston, C., Reed, B., Srivastava, U., Kumar, R., Tomkins, A.: Pig latin: a not-so-foreign language for data processing. In: SIGMOD'08: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, pp. 1099–1110. ACM, New York (2008)
27. Talasila, S., Pavan, K.I.: A generalized cloud storage architecture with backup technology for any cloud providers. Int. J. Comput. Appl. **2**(2), 256–263 (2012)