CHAPTER 8

Availability for Azure SQL

You have now made the journey from deploy to configure to security and then to monitor and tune performance. The final piece of the core of Azure SQL is availability. I have rarely talked to a SQL Server customer over the years that didn’t need their database to be highly available. I also almost never talk to a customer who doesn’t care about being able to recover from a disaster. Therefore, this chapter is really about High Availability and Disaster Recovery (HADR) for Azure SQL. I will tell you that in my opinion after now assessing and using Azure SQL over this last year, the built-in capabilities for HADR for Azure SQL are one of the great stories of the service. In fact, I believe after you go through this chapter you will be convinced that looking at Azure SQL as a target for your deployment is worth it just on the HADR capabilities alone.

In this chapter, we will spend the majority of time diving into the details of HADR capabilities including Backup and Restore, built-in HA, Azure capabilities to go farther, and database availability and consistency. Then I’ll finish the chapter on how to monitor HADR for your deployment.

This chapter will contain examples for you to try out and use as you read along. For you to try out any of the techniques, commands, or examples I use in this chapter, you will need

- An Azure subscription.
- A minimum of Contributor role access to the Azure subscription. You can read more about Azure built-in roles at https://docs.microsoft.com/en-us/azure/role-based-access-control/built-in-roles.
- Access to the Azure Portal (web or Windows application).
- A deployment of an Azure SQL Managed Instance and/or an Azure SQL Database as I did in Chapter 4. The Azure SQL Database I deployed uses the AdventureWorks sample which will be required to use some of the examples.
• To connect to Managed Instance, you will need a jumpbox or virtual machine in Azure to connect. I showed you how to do this in Chapter 4 of the book. One simple way to do this is to create a new Azure Virtual Machine and deploy it to the same virtual network as the Managed Instance (you will use a different subnet than the Managed Instance).

• To connect to Azure SQL Database, I’m going to use the Azure VM I deployed in Chapter 3, called bwsql2019, and configured for a private endpoint in Chapter 6 (you could use another method as long as you can connect to the Azure SQL Database).

• Installation of the az CLI (see https://docs.microsoft.com/en-us/cli/azure/install-azure-cli?view=azure-cli-latest for more details). You can also use the Azure Cloud Shell instead since az is already installed. You can read more about the Azure Cloud Shell at https://azure.microsoft.com/en-us/features/cloud-shell/.

• Installation of Azure PowerShell. Use the following documentation on how to install Azure PowerShell for your client: https://docs.microsoft.com/en-us/PowerShell/azure/install-az-ps. I installed Azure PowerShell in my Azure VM.

• You will run some T-SQL in this chapter, so install a tool like SQL Server Management Studio (SSMS) at https://docs.microsoft.com/en-us/sql/ssms/download-sql-server-management-studio-ssms?view=sql-server-ver15. You can also use Azure Data Studio at https://docs.microsoft.com/en-us/sql/azure-data-studio/download-azure-data-studio?view=sql-server-ver15. I installed both SSMS and ADS in the bwsql2019 Azure Virtual Machine.

• For this chapter, I have script files you can use for a few of the examples. You can find these scripts in the ch8_availability folder for the source files included for the book. I will also use the very popular tool ostress.exe for exercises in this chapter which comes with the RML Utilities. You can download RML from www.microsoft.com/en-us/download/details.aspx?id=4511. Make sure to put the folder where RML gets installed in your system path (which is by default C:\Program Files\Microsoft Corporation\RMLUtils).
HADR Capabilities

I want to first review with you the amazing HADR capabilities that come with Azure SQL before we dive deeper with examples into each topic.

Automatic Backups and Point-In-Time restore

Azure SQL is SQL Server, so the full complement of BACKUP and RESTORE functionality is possible. However, the promise of PaaS is to provide managed capabilities. Therefore, Azure SQL provides an automated backup system for both Managed Instance and Databases to meet your Recovery Point Object (RPO) and historical data needs. In fact, for Azure SQL Database, you are completely abstracted from the BACKUP T-SQL statement. Managed Instance will allow a COPY_ONLY backup to Azure Storage.

All backups from Azure SQL are kept on separate storage from your database and log files with automated geo-redundant mirrors. Azure SQL also offers a long-term backup retention option.

Azure SQL will use full, differential, and log backups supporting a complete Point-In-Time restore interface. In addition, you have these restore capabilities:

- Restore deleted databases.
- Managed Instance supports the RESTORE T-SQL statement from Azure Blob Storage which could be from an on-premises SQL Server backup or a COPY_ONLY backup from a Managed Instance.

Built-In High Availability

You may be used to using an Always On Failover Cluster Instance (FCI) or Always On Availability Group (AG) with SQL Server to give you high availability and achieve a desired Recovery Time Object (RTO).

As part of every Azure SQL deployment, you get a complete built-in High Availability system, just by deploying Azure SQL. This is included in your Azure subscription and fees for your deployment.

As you will see in this chapter, a General Purpose deployment will behave similar to FCI and Business Critical will be similar to AGs. Hyperscale uses a unique architecture that will feel like a combination of both. In all cases, the power of the Azure Service Fabric is used for automatic failover capabilities.
Azure Redundancy

Why rely on a single data center when you have three? Azure SQL can integrate with a capability called **Availability Zones in Azure**. Each zone is a set of one or more datacenters (so actually more than three) within an Azure region that has independent power, cooling, and networking. Azure SQL can deploy a high available solution across zones to provide you even more availability should there a failure in a particular data center.

All Azure SQL deployments are created as part of an **Azure Availability Set** which includes using different fault and update domains. Fault domains define the group of virtual machines that share a common power source and network switch. Update domains are groups of virtual machines and underlying physical hardware that can be rebooted at the same time. Only one update domain is rebooted at a time. A rebooted update domain is given 30 minutes to recover before maintenance is initiated on a different update domain. You can read more about this concept at [https://docs.microsoft.com/en-us/azure/virtual-machines/windows/manage-availability#configure-multiple-virtual-machines-in-an-availability-set-for-redundancy](https://docs.microsoft.com/en-us/azure/virtual-machines/windows/manage-availability#configure-multiple-virtual-machines-in-an-availability-set-for-redundancy).

Geo-replication and Auto-failover Groups

You might want to provide even higher levels of availability by synchronizing your deployments across Azure regions. Azure SQL provides two methods for this capability called Geo-replication and Auto-failover groups. We will go into each option in this chapter and why you may want to choose one vs. the other.

Database Availability and Consistency

With SQL Server, you are used to using various techniques to make your database available and checking consistency. Azure SQL eliminates the need for heavy “emergency” recovery options and provides many built-in consistency checks. I’ll explore more in this chapter specific comparisons for database availability, recovery, and consistency for Azure SQL as compared to SQL Server.
SQL Server Replication

SQL Server Replication has been a popular method to provide a level of availability and database synchronization over many SQL Server releases. I won’t dive into the details of using SQL Server Replication in this chapter, but point out these capabilities:

- Azure SQL Managed Instance gives you the full capabilities to set up a transaction or snapshot replication system including publisher, distribution, and subscribers. A subscriber can be another Managed Instance database, an Azure SQL Database, or even a SQL Server in Azure VM or on-premises. Read more about Azure SQL Managed Instance and Replication at https://docs.microsoft.com/en-us/azure/azure-sql/managed-instance/replication-transactional-overview.

- An Azure SQL Database can be a subscriber from an on-premises SQL Server, SQL Server in Azure VM, or Managed Instance for transaction and snapshot replication. This might be an interesting migration option when moving to Azure SQL Database because it can provide a type of “online” migration strategy. Read more at https://docs.microsoft.com/en-us/azure/azure-sql/database/replication-to-sql-database.

Backup and Restore

Imagine you needed to set up an automated backup system for your SQL Server deployment. You basically want a system to abstract even other DBAs from worrying about performing backups. You want these backups to run regularly; use a combination of full, differential, and log backups; and be placed on storage separate from your database for full protection. And, you would also like the storage for your backups to be mirrored even across data centers in your company.

Guess what? When you deploy an Azure SQL Managed Instance or Azure SQL Database, we just do all of this by default and more. Let’s look at various aspects to the automated backup system and how to use restore with these backups.
Automatic Backups

When you deploy an Azure SQL Database or create a new database for an Azure SQL Managed Instance deployment, we monitor this activity and kick in the following schedule of backup activity:

- A full database backup once a week.
- A differential backup every 12 hours.
- A transaction log backup every 5–10 minutes. The actual frequency of log backups is based on number of vCores and database activity.

**Note**  We may vary the implementation of how we do this. The concept is that provided we give you Point-In-Time restore and meet your RPO objective.

All backups are done using standard T-SQL statements in the background and stored separately from your data and log files (even if they are on Azure Storage). In fact, your backup files are stored on Azure Storage using RA-GRS. RA-GRS stands for **read-access geo-zone-redundant storage**. This means that backup files are copied across three Azure availability zones in the primary region and also copied asynchronously to a single physical location in a different region. Read more about RA-GRS at [https://docs.microsoft.com/en-us/azure/storage/common/storage-redundancy#redundancy-in-a-secondary-region](https://docs.microsoft.com/en-us/azure/storage/common/storage-redundancy#redundancy-in-a-secondary-region).

When you deploy or create a new database, we schedule a full database backup almost immediately. We perform integrity checks on your backups using CHECKSUM and restore techniques. Read the complete story of automated backups for Azure SQL at [https://docs.microsoft.com/en-us/azure/azure-sql/database/automated-backups-overview](https://docs.microsoft.com/en-us/azure/azure-sql/database/automated-backups-overview).

Backup Retention

By default, we keep enough backup files to allow you to perform a Point-In-Time restore (PITR) within the last 7 days at any point in time. For Azure SQL Database, you can change this retention lower to 1 day or up to 35 days. This is called **short-term backup retention**. You have the same option for any database created for a Managed Instance.
Note You cannot configure the retention of 7 days for Hyperscale deployment. We are looking in the future to allow this.

Retention policies affect how far back you can restore from a point in time but also how much storage space your backups consume. You can configure the short-term retention policy for Azure SQL Database backups through the Azure portal (I’ll show you an example when I discuss long-term retentions later in this section of the chapter), az CLI (az sql db ltr-policy), or PowerShell (Set-AzSqlDatabaseBackupShortTermRetentionPolicy).

Note Anytime there is az CLI or PowerShell support, there is also REST API support because that is what az and PowerShell use. For backup retention, you can read about REST API support at https://docs.microsoft.com/en-us/rest/api/sql/backupshorttermretentionpolicies.

Short-term backup retention can be configured for databases for Managed Instance using the Azure portal as seen in Figure 8-1 for one of the databases I deployed on my Managed Instance.

Figure 8-1. Configuring short-term backup retention for a Managed Instance database
The portal only allows configuration on a database level, so for the instance you may want to use scripts for automation. Therefore, you can also manage the short-term retention policy for Managed Instance backups with az CLI (az sql midb short-term-retention-policy) and PowerShell (Set-AzSqlInstanceDatabaseBackupShortTermRetentionPolicy).

**Backup Storage Consumption and Costs**

As part of your deployment, you get free storage for backups equivalent to the maximum size of your database or Managed Instance storage size. This includes the space for all full, differential, and log backups. Even though we compress all backups, the size you need will depend on the size of your data, how many changes you make (affects size of differential and log backups), and your number of backup retention days.

If you exceed the backup storage you get for free with your managed instance maximum storage size or maximum Azure SQL Database size, you can incur extra costs for backups.

In most cases, we have found customers that use the default retention period of 7 days rarely incur extra charges. For Azure SQL, you can track if you are using extra space that is being charged on your subscription by using the Azure portal and viewing billing information with your subscription. Learn more at https://docs.microsoft.com/en-us/azure/azure-sql/database/automated-backups-overview?tabs=single-database#storage-costs.

For Azure SQL Database, Azure Metrics allow you to track and even get alerts on backup storage consumed for full, differential, and log backups. Figure 8-2 shows an example of using Azure Metrics through the portal to see what is available.
Here are some tips to help you on consuming backup storage space:

- Reduce the retention period to the shortest possible number of days per your requirements.
- The larger modifications you make (e.g., index rebuilds), the larger the space needed for differential and log backups. Take a look to ensure these operations are all needed.
- It is possible that you can increase your maximum storage size so you get more backup space, but the increase in storage size could cost less than backup storage costs.

**Point-In-Time restore**

Now that you know about the automated backup strategy we use, you may have a need to use backups to perform a restore. In some cases for SQL Server, you may run into a situation where an accident has occurred that affects availability like a database owner dropping a table.
Since we deploy a combination of full, differential, and log backups, we allow you to pick a point in time and restore back to that state using these backups. This concept is called **Point-In-Time restore (PITR)** and is available in SQL Server (provided you deploy the right backup strategy).

If you look at the SQL Server documentation at https://docs.microsoft.com/en-us/sql/relational-databases/backup-restore/restore-a-sql-server-database-to-a-point-intime-full-recovery-model, the method to restore to a point in time is to use a series of backups to restore from a log backup using the T-SQL RESTORE statement. For Azure SQL, PITR is supported for the automated backups we create for Azure SQL Managed Instance and Databases. Therefore, to perform a Point-In-Time restore, you must use Azure interfaces such as the portal, az CLI, or PowerShell.

---

**Note**  Even though the RESTORE statement is supported for a Managed Instance, your syntax is limited and you cannot perform a Point-In-Time restore. This is only to restore full backups from the COPY_ONLY option or from a SQL Server.

---

Azure SQL Managed Instance supports PITR through the Azure portal (navigate to the database in the portal and select the Restore option from the command bar), az CLI (**az sql midb restore**), and PowerShell (**Restore-AzSqlInstanceDatabase**). A PITR operation is asynchronous and creates a new database (in fact a new deployment) based on the date and time you select for the restore. Your date and time choices are based on your backup day retention.

Azure SQL Database also supports PITR through the Azure portal, az CLI (**az-sql-db-restore**), and PowerShell (**Restore-AzSqlDatabase**).

Let’s look at an example of how to use PITR for the database I deployed called **bwazuresqldb** in Chapter 4 of the book. I created this database some time back as I was writing the chapters of the book, so by now at least 7 days has gone by for a series of backups. Let’s go through an exercise where I accidentally drop a table in my database
and then use PITR to restore the database to a new database name before the drop so I can merge that data back into my current database:

1. Drop the table.

I’ll use the Azure VM **bwsql2019** I deployed in Chapter 3 and have used in the last two chapters to connect with SSMS and run the following T-SQL statement against the **bwazuresqldb** database connected as the server admin I used during deployment (remember it is based on the AdventureWorks sample):

```sql
DROP TABLE SalesLT.OrderRating;
GO
```

**Note** This is the table I created as part of an example in Chapter 7. I left around 1500 rows in the table before I dropped it. So, if we restore it correctly, we should have it back with 1500 rows.

2. Audit when the table was dropped.

I dropped the table to show the example but imagine the scenario is that someone else dropped the table and no one remembers exactly when. In order to find out when to restore the database to a point in time, we need to know when the table was dropped. If you remember, we set up auditing for this database in Chapter 6 of the book. Let’s take advantage of that auditing to see when the table was dropped. I showed you in Chapter 6 of the book how to navigate to find Audit records for a database. The records for my database just after the DROP statement look like Figure 8-3.
If I click the first BATCH_COMPLETED record, I see the DROP statement including date and time and the user like in Figure 8-4.

Figure 8-3. Audit records after a DROP statement

If I click the first BATCH_COMPLETED record, I see the DROP statement including date and time and the user like in Figure 8-4.

Figure 8-4. An audit record for a DROP statement
3. Restore the database to the time before the DROP.

I purposely waited overnight from when I dropped the table to simulate a real-world scenario where someone could realize the drop happened at an earlier point in time. Based on the preceding audit record, I know I need to restore to a point in time before 7/22/2020 at 12:02 AM UTC time. Let’s use the Azure portal to perform a restore. I’ll navigate to my database, **bwazuresqldb**, in the Azure portal and choose Restore from the command bar as seen in Figure 8-5.

![Figure 8-5. Choosing to Restore a database with PITR](image)

Notice in the working pane for this database, you can see what the earliest time you can restore.

**Note**  The earliest point in time is based on the first transaction log backup after the database is created. A full backup must first complete before the log backup is taken.

After I select Restore, I’m presented with a screen where I can fill out the time I want to restore to as seen in Figure 8-6.
Figure 8-6. Restoring to a point in time for Azure SQL Database
You will notice where I can select a date and time and then I get options for Database name, whether I want this to be part of an elastic pool, and compute and storage options (e.g., service tier). You wonder why you can choose compute and storage? This is because a restore will create a new database and you can choose that database to have the same options (General Purpose, Serverless, Business Critical, Hyperscale, vCores, etc.) as when you create a new database or different ones. I clicked **Review + create** and a new deployment was started. You need to set some expectations on time here to deploy. This is not just a new database. We must do all the things to deploy a new database and restore a full database backup and a series of differential and log backups up to the time you selected. For me, this restore took around 10 minutes. I can do all the activities we talked about in Chapter 4 of the book to look at the activity log and see how the restore deployed a new database. I recommend customers try out various scenarios here to see expected recovery times for their workload. Just remember that higher RTO and RPO objective can be achieved with concepts like geo-replication and auto-failover groups which will be discussed later in this chapter.

**Note**  Your recovery time will depend on what we need to restore to get you to the desired point in time. Learn more at [https://docs.microsoft.com/en-us/azure/azure-sql/database/recovery-using-backups#recovery-time including limits on the number of concurrent restores.](https://docs.microsoft.com/en-us/azure/azure-sql/database/recovery-using-backups#recovery-time)

4. Verify the new database has the dropped table.

   I went back to my Azure VM, **bwsql2019**, where I had connected as an admin to the logical master. Object Explorer in SSMS now shows the new database as in Figure 8-7.
If I navigate to this database and run the following T-SQL statement:

```sql
SELECT COUNT(*) FROM SalesLT.OrderRating;
GO
```

I get back 1500 rows. I now have two choices:

- Drop `bwazuresqldb` and rename this new database so my applications would lose any changes since the time for PITR.
- Merge the data from this new database from `SalesLT.OrderRating` into my original database. I realize this may not be a simple operation if foreign keys exist.

**Long-Term Retention Backups**

You have seen we can keep backups for your databases from 1 to 35 days to meet your RPO requirements. However, what if your business needs to meet certain regulations and keep a longer history of backups? You may not need these backups to recover from a
disaster, but rather need them to access historical data to meet some type of compliance. Azure SQL offers a concept called **long-term backup retention (LTR)**. This allows you to keep backups for up to **10 years**. All LTR backups that are kept are full database backups.

---

**Note** At the time of the writing of this book, Azure SQL Managed Instance offers LTR in a limited preview, and you must use PowerShell to configure LTR. See this documentation page [https://docs.microsoft.com/en-us/azure/azure-sql/managed-instance/long-term-backup-retention-configure](https://docs.microsoft.com/en-us/azure/azure-sql/managed-instance/long-term-backup-retention-configure) for more details. You could as an alternative use the COPY_ONLY backup feature of Managed Instance to create any schedule for backups to your Azure Storage account for any period of time you like.

---

Here is how the concept works. You configure a retention period > the maximum of short-term retention of 35 days. You do this by selecting the frequency of how long you want LTR backups to be kept on a weekly, monthly, and yearly basis. You can select all three options if you like. Azure SQL will take short-term retention backups and copy them to a different Azure storage account (it also uses RA-GRS so is geo-redundant) based on the choices you make. LTR is not included as part of the subscription fee for your database but is charged a less expensive rate than excess short-term backup retention storage over what is included with your deployment.

Let’s look at one of my databases and see an example of how you could configure LTR. In order to configure LTR in the Azure Portal, I need to navigate to my logical server vs. directly to the database. Figure 8-8 shows how I can select the Manage Backups from the Resource menu of the logical server, select a database I want to configure, and select the Configure retention option.
When I select Configure retention, I’m presented with a screen where I can configure short-term backup retention and LTR. Figure 8-9 shows the choices you can make.

Figure 8-8. Configuring backup retention for Azure SQL Database

Figure 8-9. Configuring backup retention policies for Azure SQL Database
Let me explain how the choices I’ve made affect backup retention for this database:

- I modified the short-term backup retention for the database to 35 days.
- I chose that each full database backup taken weekly be kept for six weeks.
- I chose that the first full backup taken for a month be kept for 12 months.
- I chose that the full backup taken on the 16th week of the year be kept for 10 years.

You don’t have to choose all three options as I did. You can choose various combinations or just one of them. We have a chart in our documentation that can help you sketch out a schedule for LTR at [https://docs.microsoft.com/en-us/azure/azure-sql/database/long-term-retention-overview#how-long-term-retention-works](https://docs.microsoft.com/en-us/azure/azure-sql/database/long-term-retention-overview#how-long-term-retention-works).

You can also configure LTR for Azure SQL Database using az CLI ([azsqlldblr-policy](https://docs.microsoft.com/en-us/azure/azure-sql/database/long-term-retention-overview#how-long-term-retention-works)) or PowerShell ([Set-AzSqlDatabaseBackupLongTermRetentionPolicy](https://docs.microsoft.com/en-us/azure/azure-sql/database/recovery-using-backups#geo-restore)).

---

**Note**  If you use geo-replication or auto-failover groups, which I will discuss later in this chapter, you can configure LTR for those databases, but LTR backups are not taken unless that database becomes a primary.

---

**Geo-restore of Databases**

Let’s say that your database from Azure SQL Database or Managed Instance is unavailable due to a data center outage. While these situations are rare, it would be nice to able to restore a backup from a geo-redundant backup from another region that does not have an outage.

The process is called **geo-restore** and is outlined at [https://docs.microsoft.com/en-us/azure/azure-sql/database/recovery-using-backups#geo-restore](https://docs.microsoft.com/en-us/azure/azure-sql/database/recovery-using-backups#geo-restore). The concept is that you will deploy a new Azure SQL Database or create a new database for a Managed Instance based on a backup. When you select this option, you
will be presented with all the known backups that exist for your existing Azure SQL deployments. If the data center is down where your backups are normally stored, we will retrieve the geo-redundant copy of the backup from a different region.

**Restore Backups from Deleted Databases**

The built-in HADR options just keep coming with Azure SQL. Let’s say you *accidentally* delete a database for an Azure SQL Database or Managed Instance.

---

**Note**  Deleting a database for Azure SQL Database or Managed Instance does more behind the scenes than a traditional SQL Server given we have all types of services and operations tied to the database. This is why you can delete a database through Azure interfaces or the DROP DATABASE T-SQL statement.

---

I realize this may not be something you almost never see, but one of the great stories for built-in HADR is that when you delete a database, you can restore from backups associated with the deleted database using PITR based on the retention period you configured. If your retention period is 7 days and you delete a database, we can’t perform any more backups, but you can restore to a point in time from the earliest backup point to the time of database deletion.

---

**Note**  You cannot recover from a deleted logical server or Managed Instance. However, if you have configured LTR backup retention, you can create new databases based on these backups. You may be asking since LTR backups are not free, how do I ever remove LTR backups? You can use the PowerShell cmdlets `Remove-AzSqlDatabaseLongTermRetentionBackup` or `Remove-AzSqlInstanceDatabaseLongTermRetentionBackup` even if the logical server or Managed Instance has been deleted.

---

You can restore deleted databases through the Azure portal, az CLI, or PowerShell. Read more about how to do this at [https://docs.microsoft.com/en-us/azure/azure-sql/database/recovery-using-backups#deleted-database-restore](https://docs.microsoft.com/en-us/azure/azure-sql/database/recovery-using-backups#deleted-database-restore).
I’ve mentioned that it is possible to execute a BACKUP T-SQL statement for an Azure SQL Managed Instance earlier in this chapter. We have referred this capability as a native database backup. The term native is used because you can perform a full database backup using the BACKUP T-SQL statement to disk storage. This disk storage must be an Azure storage account and uses the “backup to URL” capability that SQL Server has supported for several releases.

You must use the WITH COPY_ONLY option to back up a database for a Managed Instance. SQL Server has supported a copy-only backup for several releases. A copy-only backup does not affect the backup sequence of full, differential, and log backups. Our team has posted a nice blog about how to set up a native backup with Managed Instance at https://techcommunity.microsoft.com/t5/azure-sql-database/native-database-backup-in-azure-sql-managed-instance/ba-p/386154.

Since you can perform a native backup with a Managed Instance, you can also restore these backups using the T-SQL RESTORE statement. You can only restore copy-only backups from a Managed Instance to an existing or new Managed Instance. Just remember it is a new database. For example, you will get an error (Mg 41901) if you try to use the WITH REPLACE syntax of RESTORE. You cannot restore these backups to SQL Server or Azure SQL Database.

**Note** The reason you cannot restore a backup from Managed Instance to a SQL Server is because Managed Instance is versionless.

In addition, you can take any full database backup from any supported version of SQL Server and restore it to a Managed Instance. This is actually the process you use to perform an offline or online migration from SQL Server to a Managed Instance. Learn more at https://docs.microsoft.com/en-us/azure/dms/tutorial-sql-server-managed-instance-online.

The following documentation page walks you through the process of performing a native restore with a Managed Instance: https://docs.microsoft.com/en-us/azure/azure-sql/managed-instance/restore-sample-database-quickstart.
**Built-In High Availability**

SQL Server has a great tradition of providing the necessary capabilities and software to keep your database highly available. The tradition started with Always On Failover Cluster Instance (FCI) using shared storage and integrating with technologies such as Windows Server Failover Clustering (WSFC) for automated failover. In SQL Server 2012, we introduced Always On Availability Groups to allow a non-shared storage approach, read replicas, and still integrated with WSFC for automatic failover decisions. SQL Server Linux also supports these capabilities but is integrated with Linux technologies such as Pacemaker.

One of the aspects of Azure SQL the engineering wanted to provide was a public commitment to a Service-Level Agreement (SLA) included availability. They also wanted the deployment process for a database or Managed Instance to “just do it” when it came to configuring and setting up availability. Finally, since Azure SQL was deployed with Azure Service Fabric (SF), we needed to integrate with SF for failover decisions.

The result is truly an amazing story. Every Azure SQL deployment option you choose has a built-in High Availability. Let’s take a look at each deployment option and the architecture of availability that makes it all happen. You can also use the following documentation page as a reference: [https://docs.microsoft.com/en-us/azure/azure-sql/database/high-availability-sla](https://docs.microsoft.com/en-us/azure/azure-sql/database/high-availability-sla). The details of this chapter for General Purpose and Business Critical service tiers apply to both Azure SQL Managed Instance and Database.

I asked Girish Mittur Venkataramanappa, a Principal Group Software Engineering Manager, who has worked on SQL availability for many years on the importance of our investment in built-in availability for Azure. According to Girish, “Our customers are increasingly migrating mission critical workloads to SQL DB. Availability outages not only impact our customer’s business, operations, and bottom line, it also erodes their own customer trust putting their business at risk. At Microsoft, our mission is to empower every person and every organization on the planet to achieve more. We offer strong Availability SLAs and stand behind them. We have built very effective High Availability technologies that offer redundant stand by replicas, multiple Availability Zones, and GeoDR which protect Databases against a variety of failures. In addition we have sophisticated monitoring, alerting and self-healing capabilities, and a well-trained 24x7 live site DevOps team. We take Availability very seriously. Most recently we invested a couple of years reengineering our Database Crash Recovery algorithms so
that crash recovery is near instant as opposed to several minutes prior to that. We have a continuous development cycle of measuring and improving availability for the millions of Databases that we run today.

**General Purpose High Availability**

I’ve described the overall concept of a General Purpose (GP) service tier several times in this book. Your databases are stored on Azure Storage, while tempdb is stored on local SSD storage. Let’s use a visual to describe more about the General Purpose architecture and how availability and failover work. (these figures are based on diagrams in the documentation at https://docs.microsoft.com/en-us/azure/azure-sql/database/high-availability-sla#basic-standard-and-general-purpose-service-tier-availability).

First, let’s look at Figure 8-10.

![Figure 8-10. Application connects to a General Purpose deployment using Gateways](image-url)
An application will connect to the primary replica (there is only one replica with General Purpose) by using Gateways in the control ring in an Azure region. If you remember, we talked about connection types of proxy and redirect with gateways in Chapters 4 and 6 of the book. In either case, the gateways provide a connection abstraction for the application. Notice the local SSD storage for the deployment is where tempdb is stored.

Figure 8-11 shows the next piece of the architecture.

![Azure Region](image)

**Figure 8-11. Storage for a General Purpose deployment**

The Database and transaction log files are put on Azure premium storage using LRS (Locally Redundant Storage). This means your database and transaction log files are replicated three times within a physical location of the data center. You can read more about LRS at [https://docs.microsoft.com/en-us/azure/storage/common/storage-redundancy#redundancy-in-the-primary-region](https://docs.microsoft.com/en-us/azure/storage/common/storage-redundancy#redundancy-in-the-primary-region).

As I’ve described earlier in this chapter, your backup files are stored in a different storage location within the data center but using RA-GRS so they are geo-redundant.

The Azure SQL deployment is integrated with Service Fabric (SF) to detect problems (e.g., a node failure) and initiate a failover if necessary. If a failover is required, we will find a new node with spare capacity to host your deployment as seen in Figure 8-12.
The local storage of the new node hosts SQL Server including tempdb. The new SQL Server will be directed to your database and log files on Azure storage. This type of architecture is very similar to how an FCI works for SQL Server. Your downtime is based on how fast we can find a new node with enough capacity to host your deployment choices (vCores, etc.). In addition, SQL Server that is hosting your database on the new node has just started up with a cold buffer and plan cache, so normal startup activities will affect your performance (recovery of the database will be extremely fast since we use Accelerated Database Recovery).

You may be wondering how does the application connect to the new node after a failover? The gateways are the answer. The application never changes any names to connect to the new node. The gateways take care of that logic. The application must simply retry a connection and is off and running.

You can learn more about the GP availability architecture at https://docs.microsoft.com/en-us/azure/azure-sql/database/high-availability-sla#basic-standard-and-general-purpose-service-tier-availability.
Note The Serverless compute tier uses the same high availability architecture since today it is only available for General Purpose service tiers. Some HADR features such as Long-Term Backup Retention will prevent the autopause feature of Serverless to be used. Learn more at https://docs.microsoft.com/en-us/azure/azure-sql/database/serverless-tier-overview#autopausing-and-autoresuming.

Business Critical High Availability

A Business Critical (BC) deployment relies on local storage and a series of replicas, much like an Always On Availability Group (AG). Let’s look at the BC architecture as compared to General Purpose. Figure 8–13 shows the basic architecture of a BC deployment.

Figure 8–13. A Business Critical deployment with replicas
You can see from this figure that gateways are still an important part of connectivity and that a primary replica exists. Local storage is used for tempdb but also for database and log files. In addition, like an Always On Availability Group, there are secondary replicas. For a BC deployment, we always keep four replicas up and running (one primary and three secondaries). From a transaction point of view, a commit cannot proceed on the primary replica until \textit{at least one} of the secondary replicas has acknowledged the changes are hardened.

You can also see that backup files are stored in Azure storage with RA-GRS just like a General Purpose deployment.

If a failover is necessary, we simply need to choose a secondary replica that is synchronized and make that the primary replica just like an AG as seen in Figure 8-14.

\textbf{Figure 8-14. A failover for a Business Critical deployment}

Downtime is significantly less General Purpose since the new primary replica simply has to run undo recovery to become available. Since Accelerated Database Recovery (ADR) is enabled, undo recovery can be very fast. I’ll discuss more about the important of ADR later in this chapter. You can read the entire Business Critical high availability
story at https://docs.microsoft.com/en-us/azure/azure-sql/database/high-availability-sla#premium-and-business-critical-service-tier-availability.

If the old primary replica is not usable, we will need to spin up a new secondary replica (and synchronize it) to keep four available.

Just like General Purpose service tier, applications just need to reconnect and start running again due to the use of gateways. In addition, a BC deployment will allow you to use one of the secondary replicas as a read-only replica as part of your monthly free for using BC. Our gateways help provide the redirection logic. You simply just ensure application supplied the correct “read intent” option. You can learn more at https://docs.microsoft.com/en-us/azure/azure-sql/database/read-scale-out.

---

**Note**  
Read Scale-Out supports session-level consistency. It means if the read-only session reconnects after a connection error caused by replica unavailability, it may be redirected to a replica that is not 100% up to date with the read-write replica. Likewise, if an application writes data using a read-write session and immediately reads it using a read-only session, it is possible that the latest updates are not immediately visible on the replica. The latency is caused by an asynchronous transaction log redo operation.

---

Let’s take a look at an example to show you that even though GP provides good built-in availability to your application, BC can truly be faster and provide higher availability.

For this example, I’m going to use the same General Purpose database I deployed in Chapter 4 and have used in other chapters called bwazuresqldb. I deployed a BC database in Chapter 4, but it was empty. So I deleted this database and deployed a new Business Critical database called bwazuresqldbbc using 8 vCores and the sample AdventureWorks database (I used 8 vCores because bwazuresqldb was scaled to 8 vCores in Chapter 7 of the book).

You will also need the ostress.exe program I’ve used in previous chapters as described and in the prerequisites in the beginning of the chapter. As I also stated at the beginning of this chapter, you will need Azure PowerShell.

I also have scripts you will use in the ch8_availability\gp_vs_bc folder.
For a client, I’ll use the Azure VM **bwsql2019** which is already set up to connect to the logical server for these databases:

1. Log in to Azure with PowerShell and set your subscription context. I first had to log in to Azure with PowerShell using the command

   ```powershell
   Connect-AzAccount
   ```

   This prompts me for MFA which is required at Microsoft.

   I then need to set the context to my correct subscription with these commands

   ```powershell
   Set-AzContext -SubscriptionId $subscriptionId
   ```

   where $subscriptionId is set to the subscription associated with the resource group of the database deployment.

2. Prepare and run scripts to test General Purpose availability.

   In order to see these results correctly, I recommend you have two PowerShell windows displayed side by side.

   In the left-hand window, you will want to run the script **querybase_gp.cmd**. This script looks like the following:

   ```cmd
   ostress.exe -S<server>.database.windows.net -Q"SELECT COUNT(*) FROM SalesLT.Customer" -Uthewandog -dbwazuresqldb -P<password> -n1 -r50000
   ```

   You should substitute in your server name, login, database, and password for your deployment. You can see this script uses ostress to find the count for a table in the database with a single user over 50,000 iterations. The idea here is to query the table over and over. ostress.exe is smart enough to retry a connection should a failover occur.

   In the right-hand window, you will be running the script **failoverbase_gp.ps1** which looks like the following:

   ```powershell
   $resourceGroup = "<resource group>"
   $server = "<server>"
   $database = "<database>"
   ```
Invoke-AzSqlDatabaseFailover -ResourceGroupName $resourceGroup -ServerName $server -DatabaseName $database

You will substitute in your resource group, server, and database. Note the server is not the fully qualified DNS name, just the logical server name.

Now execute the script in the left-hand window `querybase_gp.cmd`. You should see results scroll across your screen like this:

```
----------
847
(1 row affected)
----------
847
(1 row affected)
```

We are just repeating this query over and over.

Now run in the right-hand window the script `failoverbase_gp.ps1`. Very quickly in the left-hand window, you will notice errors like these:

```
07/22/20 21:44:17.444 [0x00001E6C] Attempt to establish connection failed. See the detailed errors that follow:
07/22/20 21:44:17.444 [0x00001E6C] SQLState: HY000, Native Error: 40613
[SQL Server]Database 'bwazuresqldb' on server 'bwazuresqlserver' is not currently available. Please retry the connection later. If the problem persists, contact customer support, and provide them the session tracing ID of '{CC39135B-D638-4A51-BB25-EABB8A5315A0}'.
```

Then within about 30 seconds, you will see the count of rows appear again. This shows a failover occurred and the application was down for a short period of time but then can reconnect and just get the same results. You can exit the script by hitting <ctrl>+<c> and typing in “Y” to quit.
We only allow manual failovers for Azure SQL Database (except Hyperscale and is not allowed for Managed Instance) and only allow them every 30 minutes. A manual failover requires a lot of things in the background to ensure you are available, so if we allowed you to do this anytime you want as much as you want, it could overwhelm our infrastructure systems. Therefore, if you tried to run failoverbase_gp.ps1 immediately again, you will get this error after a few seconds:

Invoke-AzSqlDatabaseFailover : Long running operation failed with status 'Failed'. Additional Info:'There was a recent failover on the database or pool if database belongs in an elastic pool. At least 30 minutes must pass between database failovers.'

You can read more about testing high availability at https://docs.microsoft.com/en-us/azure/azure-sql/database/high-availability-sla#testing-application-fault-resiliency.

3. Test failover for Business Critical.

Perform the same tests as with General Purpose, but this time use the scripts querybase_bc.cmd and failoverbase_bc.ps1. You will this time substitute all the same information except the database should be your Business Critical database (mine was called bwazursqldbbc).

Then using the same concept as in step #2, run the script to query the database and then run the script to invoke a failover. See any differences? The result set of rows should come back in seconds with a Business Critical failover.

This demonstrates both General Purpose and Business Critical have built-in availability, but Business Critical provides the highest level of availability.
Hyperscale High Availability

I’ve described the unique characteristics of the Hyperscale service tier for Azure SQL Database in several chapters of the book so far. Let’s dive further into the pieces of the Hyperscale architecture including the interesting way availability is handled. For a complete reading on Hyperscale high availability, see the documentation at https://docs.microsoft.com/en-us/azure/azure-sql/database/high-availability-sla#hyperscale-service-tier-availability.

First, let’s look at how compute nodes and replicas are part of the architecture as in Figure 8-15.

Figure 8-15. Compute nodes for Hyperscale

The primary compute node is a primary replica for a Hyperscale deployment. Hyperscale has zero to four secondary replicas which are represented as secondary compute nodes. I’ll discuss more about how replicas work shortly. The primary
compute node hosts a SQL Server for your database. This SQL Server has the standard components like a buffer pool to host database pages. Additionally on this primary compute node is your tempdb databases on local storage plus an RBEX cache. Caches in Hyperscale are all files on local SSD storage. RBEX stands for **Resilient Buffer Pool Extension**. It is similar but not exactly the same as the Buffer Pool Extension (which you can read more about at [https://docs.microsoft.com/en-us/sql/database-engine/configure-windows/buffer-pool-extension](https://docs.microsoft.com/en-us/sql/database-engine/configure-windows/buffer-pool-extension)). The concept is that if a query needs a database page and it is not in buffer pool, it will first try to read that page from RBEX.

What if a page is not available in either the compute buffer pool or RBEX cache? Figure 8-16 shows that we deploy a set of **page servers**.

---

**Figure 8-16.** Page servers in a Hyperscale deployment
Page servers are nodes with SQL Servers that host database pages in memory and are covered by another RBEX cache. Page servers are paired for redundancy and high availability. Azure SQL determines the proper number of page servers to support the deployment and database size. Remember, for Hyperscale you don’t choose a maximum database size. We just keep growing and scaling the system through page server and Azure storage to meet your database size.

What if the database page the query needs is not on the compute node or a page server? Figure 8-17 shows that the database files that back your database are stored in Azure standard storage.

**Figure 8-17. Data files for Hyperscale on Azure storage**

In order to maximize performance, this architecture works best when we rarely have to go to Azure storage to retrieve a page for a database. When page servers first start up, they are seeded with pages from database files on Azure Storage. Page servers then will
populate RBEX caches on the primary node (and secondaries if they exist). If a page is not on the primary compute buffer pool but we find it in the RBEX cache of the node, that is considered a *cache hit*. If the page is not in RBEX, we attempt to get the page from a page server (or its RBEX cache) but is considered a *cache miss*.

Having database files on Azure storage has one major advantage for automated backups and Point-In-Time restore for Hyperscale. Because the data files are infrequently accessed once the Hyperscale caching system is *warm*, we can back up the database using **snapshot backups**. This capability is very similar to using file snapshot backups with Azure Virtual Machine as documented at https://docs.microsoft.com/en-us/sql/relational-databases/backup-restore/file-snapshot-backups-for-database-files-in-azure. Snapshot backups are a huge advantage for Hyperscale because it doesn’t affect application operations and a restore of a database is extremely fast. Just like other Azure SQL options, backups are stored on geo-redundant storage separate from the data and log files.

There is another piece I haven’t discussed yet from this model. What about the transaction log?

Hyperscale redirects a transaction log I/O from the primary node to a **Log Service** as seen in Figure 8-18.
Compute nodes still have a log cache, but when an I/O is needed to flush log records, they are directed by the SQL Server engine to the Log Service. The Log Service runs on a different node. It has its own log cache (local SSD storage), log storage with Azure Premium Storage called the landing zone, and redundant storage with Azure Standard Storage called long-term log storage. We never have to back up the transaction log. A combination of snapshot backups and long-term log storage can be used to restore to a point in time.

The Log Service receives logged changes and then feeds these changes to update page servers and replicas (if they exist). This means that while Hyperscale uses logged changes to feed replicas, it doesn’t use the exact underlying technology of Always On Availability groups to keep replicas in sync.

Figure 8-18. The Log Service in Azure SQL Database Hyperscale
One interesting aspect to Hyperscale is database pages I/O. For the compute node, dirty pages are not written to a database file. Hot pages on the compute node are written to RBEX cache so they are readily available. Page servers are updated through logged changes with the Log Service. Therefore, any type of checkpoint I/O happens from a page server to database files on Azure storage. This is very nice as it off-loads any database file I/O from the compute node.

I mentioned earlier the concept that you can have zero to four secondary replicas. You may be asking how can this architecture support built-in high availability with zero replicas? Because the underlying database and transaction log files are on Azure storage (and not on local storage), if we need to execute a failover, we simply provision a new node and data will be synchronized from underlying page servers (which are backed by Azure storage). Furthermore, like all of Azure SQL, Accelerated Database Recovery (ADR) is enabled, so when the new compute node comes online or a secondary replica becomes the primary, recovery to get to a consistent state is extremely fast. Every aspect of the Hyperscale distributed architecture is fault tolerant.

If we have secondary replicas provisioned, failover (which like other Azure SQL architectures is integrated with Azure Service Fabric) is of course must faster because we can just switch to one of those nodes to become the new primary node. In addition, if your application connects to the database with read intent, Azure SQL will load balance across all available secondary replicas. Read more about using read-scale replicas with Hyperscale at https://docs.microsoft.com/en-us/azure/azure-sql/database/service-tier-hyperscale#connect-to-a-read-scale-replica-of-a-hyperscale-database.

Want to learn more about Hyperscale? Watch my colleague Kevin Farlee demonstrate the architecture of Hyperscale including an amazing restore demonstration at www.youtube.com/watch?v=Z9AFnKI7sfI.

**Tip** Remember, because of the unique architecture of Hyperscale, you cannot change the service tier back to General Purpose or Business Critical once you deploy or change to Hyperscale. You would have to export out your data and import into the new tier.
Go Further with Azure

While the built-in high availability of Azure SQL is really a major advantage for you to consider moving to the cloud, there are ways to go even further and build in more availability to your plans.

This includes zone redundancy, geo-replication, and auto-failover groups. In addition, it is important to understand how the promise of the Azure SQL Service-Level Agreement meets your needs. This includes how we govern and limit certain activity and deploy innovative technologies like hot patching to maximize uptime.

Zone Redundancy

I mentioned in Chapter 2 of the book features of the Azure ecosystem including regions and datacenters. One of the capabilities you can take advantage of to infuse higher availability is to deploy with Azure Availability Zones. Availability zones are a collection of unique physical locations within a region. Each zone in the collections is made up of multiple data centers. Today, availability zones are available only to Azure SQL Database Business Critical service tiers and is free of charge.

---

Note  We are working to expand availability zones to other Azure SQL deployment options.

---

You can choose an availability zone during deployment or configure it later. Figure 8-19, from the documentation at https://docs.microsoft.com/en-us/azure/azure-sql/database/high-availability-sla#zone-redundant-configuration, shows a visual example of how availability zones are implemented with a Business Critical service tier deployment.
Notice that each secondary replica is deployed in a different zone. Each zone has its own set of gateways for connection. Additionally, an Azure Traffic Manager service is deployed automatically to redirect traffic to each set of gateways. Your application is completely abstracted for worrying about the traffic manager.

Let’s see how to configure an existing database to be zone redundant (if you remember back to Chapter 4 this was a choice I could have made when creating a business critical tier database but I did not enable it).

Figure 8-19. An availability zone implementation with an Azure SQL Database Business Critical deployment
If I navigate to my database `bwazuresqldbbc` and select Configure from the Resource menu, I see an option to make my database zone redundant as seen in Figure 8-20.

![Configure the database zone redundancy](image)

**Figure 8-20. Making a database zone redundant**

This option may only be available in certain Azure regions. Keep up with the latest support for regions at [https://docs.microsoft.com/en-us/azure/availability-zones/az-region](https://docs.microsoft.com/en-us/azure/availability-zones/az-region).

I selected Yes on this screen and hit the Apply button. As you can imagine, moving to zones is not completely trivial as we have to redeploy your replicas across zones within the region. However, for my deployment which is a small database, the operation only took a few minutes. You can also configure zone redundancy with the az CLI (`az sql db update`) and PowerShell (`Set-AzSqlDatabase`). Availability zones are not available in all regions. Check your region availability at [https://docs.microsoft.com/en-us/azure/availability-zones/az-region](https://docs.microsoft.com/en-us/azure/availability-zones/az-region).
Note Because replicas are now in different datacenters, you may experience some increased latency for workloads, especially write-intensive OLTP applications. You will have to balance the benefit of availability with the performance needs of your application. You can always easily switch back to a single zone deployment.

You may be wondering whether you even need a zone redundant deployment. I mean how often does a datacenter have a major issue. I asked Emily Lisa, lead program for Azure SQL over Availability Zones, about her thoughts. She gave me an interesting perspective: “As I am writing this quote I am working from home at my desk after several months in quarantine due to the COVID-19 pandemic. After the world suddenly and unexpectedly shut down I had to find new ways and places to stay “available” for my job. Through this experience I’ve realized how important it is to recognize that nothing is perfect, unexpected things happen and being ready to adapt to new circumstances is crucial. At any moment, a natural disaster can take out an entire datacenter, a network failure can impact the functionality of servers, and many other unexpected events can occur potentially jeopardizing the availability of your database. We embrace this reality by actively working to minimize the effects of a single failing component with as little downtime as possible. With Azure Availability Zones, databases can be replicated within the same region across several unique physical locations with independent power, cooling, and networking. The zone redundant configuration which is currently available for Azure SQL Database Premium, Business Critical, and General Purpose (coming soon) tiers makes databases resilient to a large set of failures, including catastrophic datacenter outages, without any changes of the application logic. With this feature enabled, even a meteor can crash and destroy an entire datacenter and your database will still be up and running in a different Availability Zone within the same region (too bad the dinosaurs didn’t have a zone redundant configuration). This gives peace of mind by ensuring low downtime and no loss of committed data, while automatically handling virtually all maintenance and failovers. Zone redundancy is the future of HA in the cloud!”

Geo-replication

Let’s say you want to go even further and make your database resilient across Azure regions. Azure SQL Database supports a concept called geo-replication. Geo-replication uses Always On Availability Group technology to asynchronously transmit log changes
to another Azure SQL Database deployment on a different logical server. The secondary database can be in a different Azure region or the same Azure region. Secondary databases can be used for read replica purposes.

**Note** Using a geo-replicated database in the same region can give a General Purpose service tier a read replica or expand the number of replicas for a Business Critical service tier (which by default gets one).

Geo-replicated databases can be used for failover purposes, which include unexpected Azure region events or to support an application upgrade with minimal downtime. However, failover is a manual process initiated by an administrator through Azure interfaces or through T-SQL (ALTER DATABASE).

Let’s see how to take the Business Critical deployment I just enabled for availability zones and create a geo-replicated secondary database in another Azure region. I’ll navigate to my database, called `bwazuresqldbbcc`, in the Azure portal and select Geo-Replication from the Azure portal like in Figure 8-21.

![Figure 8-21. Configuring geo-replication for Azure SQL Database](image)
Note You see a message on this screen about using auto-failover groups. I will discuss auto-failover groups in the next section of the chapter.

I want to replicate the database in the South Central US region so will select that region. I now get a new screen to configure the new logical server to use (could be an existing server in that region) and choose a Pricing tier. You can see in Figure 8-22 I created a new server as part of the process called bwazuresqlserver2 (on that screen I was allowed to choose “Allow Azure services to access server”).

![Create secondary](image-url)  
Create secondary

Home > bwazuresqldbbcc (bwazuresqlserver/bwazuresqldbbcc) | Geo-Replication >

**Region**

South Central US

**Database name**

bwazuresqldbbcc

**Secondary type**

Readable

**Target server**

bwazuresqlserver2 (South Central US)

**Elastic pool**

None

**Pricing tier**

Business Critical: Gen5, 8 vCores, 32 GB storage

**Figure 8-22. Choices for creating a geo-replicated database**
Notice the Pricing tier matches the deployment, vCore, and storage of my current database. You do have the option of increasing or decreasing vCore capacity for the secondary, but you can’t use other service tiers. It may be tempting to use a lower capacity for the secondary, but there could be consequences for this choice including

- A secondary that gets significantly out of sync
- Transaction log governance on the primary to ensure a secondary doesn’t get too far out of sync
- An improperly sized new primary if a failover has to occur (although you could change it after a failover)

The deployment creates a new database of the same name as the primary on the new logical server and then performs an initial synchronization called **seeding**. When this operation is complete, you can see the status of the new secondary like Figure 8-23.

Any changes from my primary are asynchronously sent to the secondary. In addition, I can now connect to bwazuresqlserver2.database.windows.net and perform read operation against the bwazuresqldbc database. The database is read-only, so any modification operations would fail. Remember I also have access to a secondary read replica in my primary region since I used Business Critical. In the case of the secondary replica in the primary region, I need to use a connection string option (as documented at `https://docs.microsoft.com/en-us/azure/azure-sql/database/read-scale-out#connect-to-a-read-only- replica`). When I connect to

---

**Figure 8-23. A geo-replicated secondary database after seeding**
the geo-replicated secondary database, I don't need this special connection string option. Azure SQL supports up to four secondary databases configured with geo-replication. By implementing a geo-replication architecture for a Business Critical database, I've effectively set up a deployment similar to a distributed availability group in SQL Server (learn more at https://docs.microsoft.com/en-us/sql/database-engine/availability-groups/windows/distributed-availability-groups).

**Tip** You are allowed to create geo-secondary databases based on a geo-replicated secondary database, thus giving you even more read replica. This is called *chaining*, but know that these chained secondaries will likely have a lag of data synchronization the further you build the chain.

Applications can use Azure technology like Azure Traffic Manager to set up abstraction to connect to a primary even after failover. Learn more from our documentation about building applications for global Azure SQL Database deployments at https://docs.microsoft.com/en-us/azure/azure-sql/database/designing-cloud-solutions-for-disaster-recovery. Our documentation at https://docs.microsoft.com/en-us/azure/azure-sql/database/active-geo-replication-overview has a nice visual showing what a fully deployed Azure SQL Database with geo-replication can look like as seen in Figure 8-24.
Here are some important points to consider when using geo-replication:

- You can set up a geo-replicated database on another Azure subscription with a little work. Learn more at https://docs.microsoft.com/en-us/azure/azure-sql/database/active-geo-replication-overview#cross-subscription-geo-replication.

- Server-level firewall rules on the primary are not replicated, so consider using database firewall rules or other methods to connect to the secondary.

- Using contained database users (even with Azure Activity Directory) has a huge advantage as they are replicated.

- Geo-replication uses asynchronous replication of data. Therefore, if you fail over to a secondary, you might experience data loss (but not consistency). However, if you require a secondary to be completely in
sync before a failover, you can use the `sp_wait_for_database_copy_sync` stored procedure. Learn more at https://docs.microsoft.com/en-us/azure/azure-sql/database/active-geo-replication-overview#preventing-the-loss-of-critical-data.

- Az CLI supports creating and configuring geo-replication (e.g., `az sql db replica`), and PowerShell cmdlets exist to support geo-replication (e.g., `New-AzSqlDatabaseSecondary`).

**Auto-failover Group**

While geo-replication is a great capability, what about Azure SQL Managed Instance? Also, it would be nice to have an option where failover is automatic and for an abstraction on the connection to the primary database wherever it exists. That is a nutshell what **auto-failover groups** provide.

Figure 8-25 shows a visual of an auto-failover group so you can compare to geo-replication.

![Auto-failover groups for Azure SQL Database](image)

**Figure 8-25.** Auto-failover groups for Azure SQL Database
Notice here that unlike geo-replication, auto-failover groups operate at the logical server level (and then you place databases in the failover group).

Figure 8-26 shows the architecture for auto-failover groups for Azure SQL Managed Instance.

Figure 8-26. Auto-failover groups for Azure SQL Managed Instance

In both figures, notice an Azure Traffic Manager helps abstract the application (you need to implement this though). Let’s take the logical server I deployed in Chapter 4, bwazuresqlserver, and create a failover group with a new logical server in another region. Then I will add a database to the failover group.

I’ll first navigate to my logical server bwazuresqlserver in the Azure portal and select Failover groups from the Resource menu like in Figure 8-27.
I’ll select Add group to create the new failover group. Figure 8-28 shows my selections to create the failover group, which includes creating a new logical server in the South Central US region called bwazuresqlserversouth. I also chose to add the bwazuresqldbname General Purpose database to the group.

Figure 8-27. Creating a failover group for a logical server

Figure 8-28. Choices to create a failover group
Notice the first field is the **Failover group name**. You are going to love this aspect to auto-failover groups. This is effectively the *virtual logical server* name to connect. You specify this logical server name in your application, and you will always be connected to the primary logical server for read-write purposes. I’ll discuss shortly how to use the failover group name to connect to read replicas for secondaries.

I selected **Create** which fires off a new deployment for the failover group. When the deployment finished (which includes seeding like geo-replication), Figure 8-29 shows the update screen for the status of the failover group.

**Figure 8-29. Failover group after deployment**

If you click the failover group, you get a very nice global map visual with options to manage the failover group and connection information for both the primary server and read-only replicas like Figure 8-30.
I’ll come back to this screen shortly to look at failover.

Both servers in the failover group are configured to allow connectivity from my virtual machine **bwsql2019** I deployed in Chapter 3 and have been using in the book. The current primary server is configured using private link, and when I created the secondary server as part of the failover group deployment, I chose Allow Azure services to access server.

Using SSMS in my Azure VM, I connected to the standard logical server (**bwazuresqlserver**) and can see all databases, the failover group server name (**bwazuresqlww**), the secondary logical server directly (**bwazuresqlserversouth**), and the failover group logical server for read replica (**bwazuresqlww.secondary**). Figure 8-31 shows all these options.
I only connected to bwazuresqlserver and bwazuresqlserversouth to show you in SSMS. When I deploy an auto-failover group, it only makes sense to use the failover server and failover server secondary names.

Let's now test a failover using the failover group name. I'll use the scripts from the ch8_availability\gp_vs_bc folder and modify the querybase_gp.cmd script to use the server name bwazuresqlww.database.windows.net and leave everything else the same:

1. Run the workload.

Run the script querybase_gp.cmd from PowerShell. As before, you should see a stream of row counts like this:

----------
847

Figure 8-31. Connecting to servers for a failover group with SSMS
2. Navigate back to the Azure portal to failover.

Using the Azure portal, I’ll navigate back to the bwazuresqlserver logical server and select Failover groups from the Resource Menu. Then as you can see in Figure 8-32, I’ll select Failover and Yes.

![Figure 8-32. Failing over a failover group for Azure SQL Database](image)

3. Check the workload status.

For a short period of time, ostress will hit some connection errors such as these:

07/26/20 22:13:13.945 [0x000007E0] Attempt to establish connection failed. See the detailed errors that follow:

07/26/20 22:13:13.946 [0x000007E0] SQLState: 42000, Native Error: 40613

[SQL Server]Database 'bwazuresqldb' on server 'bwazuresqlserver' is not currently available. Please retry the connection later. If the problem persists, contact customer support, and provide them the session tracing ID of '1C2C1472-3C6D-4EE7-AB1D-073D3009409E'.

Notice the error to connect is for the actual logical server, not the failover group name.
4. Check the Azure portal.

The Azure portal shows in this map a visual of the failover action occurring to make bwazuresqlserversouth the new primary as seen in Figure 8-33.

---

**Figure 8-33. A failover in progress for a failover group**

After the failover has occurred, the portal shows the new primary as seen in Figure 8-34.

---

**Figure 8-34. Failover group status after a failover**

426
5. Check the workload status.

You will see the workload has already started showing row counts proving the application doesn’t need to change its connection properties to stay available.

Here are a few other important points about auto-failover groups for Azure SQL:

- Auto-failover groups can also be managed using the PowerShell (`Add-AzSqlDatabaseToFailoverGroup` and `Switch-AzSqlDatabaseFailoverGroup`).
- Because auto-failover groups use asynchronous replication data when a failover occurs automatically, data loss could occur. Therefore, if you require no data loss with auto-failover groups, applications can call the `sp_wait_for_database_copy_sync` procedure after committing a transaction to ensure all data is synchronized. Learn more at https://docs.microsoft.com/en-us/azure/azure-sql/database/auto-failover-group-overview?tabs=azure-PowerShell#preventing-the-loss-of-critical-data.
- One of the options you can configure with an auto-failover group is called a grace period using the `GracePeriodWithDataLossHours` parameter (the default is 1 hour). This parameter defines the time we will wait to do an automatic failover should the primary be down and we believe data loss may occur. If no data loss would occur, the automatic failover takes place immediately.

**Tip** There could be a scenario where the primary comes back online or is available even after one hour. Therefore, if you cannot accept data loss and are not using the `sp_wait_for_database_copy_sync` procedure, you may want to set the grace period to something like 24 hours.

- A great example of how to use an application with an auto-failover group is available in our documentation with a Java application at https://docs.microsoft.com/en-us/azure/azure-sql/database/geo-distributed-application-configure-tutorial.
A tutorial about how to add an Azure SQL Managed Instance to a failover group can be found at https://docs.microsoft.com/en-us/azure/azure-sql/managed-instance/failover-group-add-instance-tutorial.

The important limitations for auto-failover groups are system databases. System databases are not replicated. Therefore, any instance-level data, such as SQL Server Agent jobs, must be manually created on the secondary instance.

Auto-failover groups use geo-replication technology behind the scenes but are different. My colleague Anna Hoffman created this very nice table as seen in Figure 8-35 to compare geo-replication and auto-failover groups.

|                        | Geo-replication | Auto-failover groups |
|------------------------|-----------------|----------------------|
| Automatic failover     | No              | Yes                  |
| Fail over multiple databases simultaneously | No              | Yes                  |
| Update connection string after failover | Yes             | No                   |
| Managed instance supported | No              | Yes                  |
| Can be in same region as primary | Yes             | No                   |
| Multiple replicas      | Yes             | No                   |
| Supports read-scale    | Yes             | Yes                  |

**Figure 8-35. Geo-replication vs. auto-failover groups**

**Azure SQL SLA**

One of the advantages to deploy SQL Server with Azure SQL Managed Instance and Database is the promise of availability. Our architecture which I’ve described in this chapter helps us achieve this promise. The promise for you is in the form of a Service-Level Agreement (SLA). You can view the official SLA for Azure SQL at https://azure.microsoft.com/en-us/support/legal sla/sql-database.

The Azure SQL SLA means that Microsoft will ensure we maintain a service level or you will be eligible for a credit for your account. Service levels are stated in terms of nines. Nines are a percentage of uptime or availability you are guaranteed for your deployment.
As an example, if you deploy an Azure SQL Database with the Business Critical or Premium service tier and use Zone Redundancy, your SLA is 99.995%. If you look at a table like https://en.wikipedia.org/wiki/High_availability, you can see that 99.995% is defined as “four and a half nines” and means you could experience at maximum 26.30 minutes per year or 2.19 minutes per month of downtime. Other deployment options have different service-level promises.

If you look at our SLA documentation, downtime is defined as “…the total accumulated Deployment Minutes across all Databases in a given Microsoft Azure subscription during which the Database is unavailable. A minute is considered unavailable for a given Database if all continuous attempts by Customer to establish a connection to the Database within the minute fail.”

We also include in our SLA promises for Recovery Point Objective (RPO) and Recovery Time Objective (RTO) if you use Azure SQL Database with a Business Critical service tier and use geo-replication (or auto-failover).

---

**Note** Even though Managed Instance is not specifically listed in the SLA documentation (we are working to add it explicitly), the following SLA applies to databases for a Managed Instance. “Azure SQL Database Business Critical or Premium tiers not configured for Zone Redundant Deployments, General Purpose, Standard, or Basic tiers, or Hyperscale tier with two or more replicas have an availability guarantee of at least 99.99%.”

---

There are certain aspects to Azure SQL which allow us to make these SLA promises including but not limited to

- Built-in availability and integration with the Azure Service Fabric
- Enforcing resource limits such as log governance

**Note** There are several reasons why log governance is needed to manage PaaS services. This includes database recoverability, high availability, disaster recovery, and predictable performance. Learn more at https://azure.microsoft.com/en-us/blog/resource-governance-in-azure-sql-database/.

- Enabling database options such as Accelerated Database Recovery
One innovative technology we use in Azure SQL to maximize availability is hot patching. Hot patching allows us to patch the SQL Server engine code without restarting SQL Server. Read the amazing story of hot patching from my colleague Hans Olav Norheim at https://azure.microsoft.com/en-us/blog/hot-patching-sql-server-engine-in-azure-sql-database/.

### Database Availability and Consistency

For SQL Server, you may be familiar with features and tools to make or restrict the availability of your database or perform advanced recovery scenarios. In addition, SQL Server provides tools to ensure the database is consistent, both from a physical and logical perspective.

Azure SQL in general does not provide the same level of advanced capabilities in this area mainly because they are not needed given the high level of redundancy and availability built into the service.

Let’s examine a few of these areas, so your knowledge can be more complete when comparing to SQL Server.

### Database Availability

You may have needed with a SQL Server to change the database state with ALTER DATABASE to OFFLINE or EMERGENCY for advanced recovery scenarios. You don’t have access to use these options, but after reading about all of the built-in capabilities of Azure SQL and our SLA, you have to ask “Does it matter?” In my opinion (and believe me I’ve used these options to help customers over the years in support), the answer is no.

With Azure SQL Database and Managed Instance, while you cannot put the data in single user mode, Azure SQL Database allows you to use the option RESTRICTED_USER. Learn more at https://docs.microsoft.com/en-us/sql/t-sql/statements/alter-database-transact-sql-set-options.

In SQL Server 2005, my colleague Robert Dorr and I were in Microsoft Support and convinced the engineering team to create a simple method to connect into a “hung server.” The result was a feature called Dedicated Admin Connection (DAC). DAC is supported for Azure SQL. Learn more at https://docs.microsoft.com/en-us/sql/database-engine/configure-windows/diagnostic-connection-for-database-administrators?view=sql-server-ver15.
Accelerated Database Recovery (ADR)

In my book, *SQL Server 2019 Revealed*, I covered the amazing story of Accelerated Database Recovery (ADR). No longer will a SQL Server administrator worry about long-running database recovery times or out-of-control transaction logs. ADR is not just a feature; it is part of the Azure SQL availability story! In fact, for Azure SQL, it is just part of the engine and not really something you turn on or off. You can learn more how ADR works in our documentation at https://docs.microsoft.com/en-us/azure/azure-sql/accelerated-database-recovery or in the white paper written by our engineering team at www.microsoft.com/en-us/research/publication/constant-time-recovery-in-azure-sql-database/.

Database Consistency

All Azure SQL databases are configured using the CHECKSUM option for database consistency. One of the benefits of using a PaaS service is that our engineering team has automation to check for any inconsistencies due to issues like a checksum problem and take correction action. For example, if you are deploying a Business Critical service tier, we can issue an online automatic page repair (learn more how this works at https://docs.microsoft.com/en-us/sql/sql-server/failover-clusters/automatic-page-repair-availability-groups-database-mirroring?view=sql-server-ver15).

In addition, keep in mind these facts if you are concerned about any database inconsistency issue:

- General Purpose and Hyperscale store database and log files on Azure storage which is mirrored by default with three copies.
- Business Critical tiers have three other replicas always available with their own storage.
- Our engineering team has built-in data integrity and consistency alert monitoring in our service. If automation can’t solve the problem, we will directly notify a customer and take necessary steps to ensure data is restored and consistent. If we think we can repair a problem with no data loss, we might take this action and you never have to be notified.
• Azure SQL does support DBCC CHECKDB (but not repair option) for you to manually check your database consistency at any time.

• We’ve added checks for databases for “lost write” and “stale read” detection which we have seen in some situations occur due to an underlying I/O system issue.

Peter Carlin, a Distinguished Engineer at Microsoft who I mentioned as part of Azure SQL history in Chapter 1, has a very nice blog post outlining all the things we do to manage data integrity for your database in Azure. Read his post at https://azure.microsoft.com/en-us/blog/data-integrity-in-azure-sql-database/.

Monitoring Availability

As with any set of capabilities, you will no doubt want to monitor various aspects of availability for Azure SQL. This includes server and instance availability, database availability, backup/restore history, status of replicas, and failover reasons. In addition, since Azure SQL runs in the Azure ecosystem, knowing the status and health of Azure services in regions and datacenters can also be important.

Azure SQL provides you similar interfaces to SQL Server to monitor availability, including catalog views, Dynamic Management Views (DMV), and Extended Events (XEvent). In addition, Azure interfaces such as the Azure portal, az CLI, PowerShell, and REST provide additional capabilities to monitor the availability of your deployment.

Let’s dive into a few examples of using these interfaces and monitoring capabilities.

Instance, Server, and Database Availability

Aside from Azure service-impacting events, you can view the availability of your Azure SQL Managed Instance or Azure Database Server and databases through the Azure portal. One of the primary methods to view a possible reason for a Managed Instance or Database to not be available is by examining Resource Health through the Azure portal or REST APIs.

You can always use standard SQL Server tools such as SQL Server Management Studio to connect to a Managed Instance or Database server and check the status of these resources through the tool or T-SQL queries.
In addition, interfaces such as az CLI can show the status of Azure SQL such as

- **az sql mi list** – List the status of managed instances.
- **az sql db list** – List the status of Azure SQL Databases.

PowerShell commands can also be used to find out the availability of an Azure SQL Database such as

**Get-AzSQLDatabase** – Get all the databases on a server and their details including status.

REST APIs, although not as simple to use, can also be used to get the status of Managed Instances and Databases. The complete REST API reference is at https://docs.microsoft.com/en-us/rest/api/sql/.

---

**Note**  For a SQL Server, I often look at past ERRORLOG files or the system_health XEvent session files for service availably and health. Azure SQL Managed Instance supports these tools. However, these files are not copied to replicas, so if a failover occurs, the history of these files is lost.

---

**Backup and Restore History**

Azure SQL automatically backs up databases and transaction logs. Although standard backup history is not available, long-term backup retention history can be viewed through the Azure portal or CLI interfaces. Learn more at https://docs.microsoft.com/en-us/azure/azure-sql/database/long-term-backup-retention-configure.

Additionally, Azure SQL Managed Instance supports using XEvents to track backup history. See a blog post describing how to do this at https://techcommunity.microsoft.com/t5/azure-database-support-blog/lesson-learned-128-how-to-track-the-automated-backup-for-an/ba-p/1442355.

Any restore of a database using Point-In-Time restore results in the creation of a new database so the history of restore can be viewed as looking at the creation of a new database. All operations to create a new database can be viewed through Azure Activity Logs.
To get a global view of the status of Azure regions and datacenters, use the **Azure Status** dashboard which you can find at [https://status.azure.com](https://status.azure.com). Figure 8-36 shows an example of the Azure Status dashboard.

**Figure 8-36. The Azure status dashboard**

Azure status shows the status of all Azure services in all Azure regions. The status shows all services independent of your use of a specific service. To get notified of Azure status, you can use the RSS feed at the top of the page. In addition, you can see a complete history of Azure status through Azure status history at [https://status.azure.com/status/history](https://status.azure.com/status/history).
You can also get more information about the health of Azure services specific to your subscription through the Azure portal for a capability called **Azure Service Health**. Through Service Health, you can see current issues for Azure services, planned maintenance that could affect availability, and health history. Figure 8-37 shows an example of Service Health for my subscription.

![Dashboard > Service Health > Service Health](Image)

**Figure 8-37. Azure Service Health**

The default view shows any active incidents that could affect availability of your Azure resources in addition to incidents over the last week. If you select Health History, you can get more details of past issues for up to the last 3 months. You can select Service Health from your Azure portal in the dashboard. You can also view history of the health of a specific Azure resource, such as a database or Managed Instance. I’ll show you an example of resource health later in this section.
Replica Status

To monitor the state of replicas in Azure SQL, you can use the DMV `sys.dm_database_replica_states`. This DMV could be used for example to look at the status of replicas for a Business Critical service tier. You could also use this DMV to check the status for replicas for a deployment with geo-replication or auto-failover groups.

If you remember, I deployed in Chapter 4 of this book a Business Critical service tier database called `bwazuresqldbcb`. I can connect with SSMS to this database and run the following T-SQL statement:

```
SELECT is_primary_replica, synchronization_state_desc, synchronization_health_desc, last_received_time, last_redone_time
FROM sys.dm_database_replica_states;
GO
```

I get the following results from this database on the logical server:

| is_primary_replica | synchronization_state_desc | synchronization_health_desc | last_received_time  | last_redone_time |
|--------------------|---------------------------|----------------------------|---------------------|------------------|
| 1                  | SYNCHRONIZED              | HEALTHY                    | NULL                | NULL             |

I then connected to the same logical server with the database bwazuresqldbcb with applicationintent=readonly (to connect to a read replica) and ran the same query. The results look like this:

| is_primary_replica | synchronization_state_desc | synchronization_health_desc | last_received_time  | last_redone_time |
|--------------------|---------------------------|----------------------------|---------------------|------------------|
| 0                  | SYNCHRONIZED              | HEALTHY                    | 2020-07-28 01:22:24.813 | 2020-07-28 01:22:13.880 |

For geo-replication and auto-failover groups, there are additional DMVs to check on the status of replication between logical server and instances. This includes `sys.geo_replication_links` (run in the context of the logical master) and `sys.dm_geo_replication_link_status` (run in the context of the user database). One example of
using these DMVs is seeding (the initial sync of the geo-secondary). sys.geo_replication_links can show the state of the seeding process as it progresses and completes.

In this chapter, I configured an auto-failover group. Let’s connect to both the primary and secondary failover group server and see what these DMVs look like.

I connected to the failover group server **bwazuresqlww.database.windows.net** and ran the following T-SQL statement:

```sql
SELECT partner_server, partner_database, replication_lag_sec, replication_state_desc, role_desc
FROM sys.dm_geo_replication_link_status;
GO
```

I got the following results:

| partner_server          | partner_database | replication_lag_sec | replication_state_desc | role_desc |
|-------------------------|------------------|---------------------|------------------------|-----------|
| bwazuresqlserversouth   | bwazuresqldb     | 0                   | CATCH_UP               | PRIMARY   |

A *replication_state_desc* = CATCHUP means the servers are synchronized.

I then connected to the secondary failover group server of **bwazuresqlww. secondary.database.windows.net** and ran the same query. I got these results:

| partner_server      | partner_database | replication_lag_sec | replication_state_desc | role_desc |
|---------------------|------------------|---------------------|------------------------|-----------|
| bwazuresqlserver    | bwazuresqldb     | NULL                | CATCH_UP               | SECONDARY |

You can see this is the secondary server paired with the primary logical server. *replication_lage_sec* can be used to see if there is a delay in synchronizing the servers.

You can see the full documentation of sys.dm_geo_replication_link_status at [https://docs.microsoft.com/en-us/sql/relational-databases/system-dynamic-management-views/sys-dm-geo-replication-link-status-azure-sql-database?view=azuresqldb-current](https://docs.microsoft.com/en-us/sql/relational-databases/system-dynamic-management-views/sys-dm-geo-replication-link-status-azure-sql-database?view=azuresqldb-current).
All of these DMVs to check replica status work for both Azure SQL Managed Instance and Database. az CLI can be used to check on replica status (az sql db replica). PowerShell cmdlets exist to support checking replica status, for example, Get-AzSqlDataBaseReplicationLink.

Failover Reasons

There are various reasons why a failover can occur, planned and unplanned, for Azure SQL Managed Instance and Database. Because the reasons for failover could vary, the best method to track if a failover occurred for your database is to use Resource Health for your Managed Instance or Database. Figure 8-38 shows an example of Resource Health for one of my Azure SQL Database deployments and a health event which resulted in a failover.

![Resource health history for an Azure SQL Database](image)

**Figure 8-38.** Resource health history for an Azure SQL Database

*Note* Resource health is reported at the database level for both Azure SQL Managed Instance and Database deployments.
You can also use REST APIs to check resource health for an Azure resource. Learn more at https://docs.microsoft.com/en-us/rest/api/resourcehealth/.

**System Center Management Pack for Azure SQL**

System Center Operations Management (SCOM) packs are software modules that help a user of System Center monitor applications and services. You can learn more about SCOM packs at https://docs.microsoft.com/en-us/system-center/scom/manage-overview-management-pack. We have built SCOM packs for both Azure SQL Database and Managed Instances.

My colleague Ebru Ersan leads our team on the design of SCOM packs. You can read her blog post about SCOM packs for Managed Instance at https://techcommunity.microsoft.com/t5/sql-server/released-azure-sql-managed-instance-management-pack-7-0-22-0/ba-p/1503931. You can download the SCOM pack for Azure SQL Database at www.microsoft.com/en-us/download/details.aspx?id=38829.

**Summary**

In this chapter, you learn the amazing built-in availability capabilities for Azure SQL, including automatic backups, Point-In-Time restore, and built-in availability architectures of General Purpose, Business Critical, and Hyperscale.

You also learned how to use the power of Azure for further redundancy with Azure Availability Zones, geo-replication, and auto-failover groups. You learned how Azure provides built-in data integrity capabilities and processes and how to monitor the availability of your Azure SQL deployments.

Availability is the last part of the meat and potatoes of Azure SQL. In the next chapter, you will complete your knowledge of Azure SQL to learn capabilities not specifically related to security, performance, or availability.