


Architecting Azure applications for resiliency and availability

04/10/2019 • 27 minutes to read • Contributors 

In this article

- [Conduct a failure mode analysis](#)
- [Plan for redundancy](#)
- [Design for scalability](#)
- [Determine subscription and service requirements](#)
- [Load-balance as needed](#)
- [Implement resiliency strategies](#)
- [Ensure that availability meets SLAs](#)
- [Manage your data](#)
- [Next steps](#)

After you've developed the requirements for your application, the next step is to build resiliency and availability into it. These qualities can't be added at the end — you must design them into the architecture.

Conduct a failure mode analysis

Failure mode analysis (FMA) builds resiliency into a system by identifying possible failure points and defining how the application responds to those failures. The FMA should be part of the architecture and design phases, so failure recovery is built into the system from the beginning. The goals of an FMA are to:

- Determine what types of failures an application might experience and how the application detects those failures.
- Capture the potential effects of each type of failure and determine how the app responds.
- Plan for logging and monitoring the failure and identify recovery strategies.

Here are some examples of failure modes and detection strategies for a specific failure point — a call to an external web service:

Failure mode	Detection strategy
Service is unavailable	HTTP 5xx
Throttling	HTTP 429 (Too Many Requests)
Authentication	HTTP 401 (Unauthorized)
Slow response	Request times out

For more information about the FMA process, with specific recommendations for Azure, see [Failure mode analysis](#).

Plan for redundancy

Failures vary in scope of impact. Some hardware failures, such as a failed disk, affect a single host machine. A failed network switch could affect an entire server rack. Less common failures, such as loss of power, disrupt a whole datacenter. Rarely, an entire region becomes unavailable.

Redundancy is one way to make an application resilient. The level of redundancy depends on your business requirements — not every application needs redundancy across regions to guard against a regional outage. In general, there's a tradeoff between greater redundancy and reliability versus higher costs and complexity.

Review Azure redundancy features

Azure has a number of redundancy features at every level of failure, from an individual virtual machine (VM) to an entire region.

- **Single VMs** have an [uptime service level agreement \(SLA\)](#) provided by Azure. (The VM must use premium storage for all operating system disks and data disks.) Although you can get a higher SLA by running two or more VMs, a single VM may be reliable enough for some workloads. For production workloads, however, we recommend using two or more VMs for redundancy.
- **Availability sets** protect against localized hardware failures, such as a disk or network switch failing. VMs in an availability set are distributed across up to three *fault domains*. A fault domains defines a group of VMs that share a common power source and network switch. If a hardware failure affects one fault domain, network traffic is routed to VMs in the other fault domains. For more information about availability sets, see [Manage the availability of Windows virtual machines in Azure](#).
- **Availability Zones** are physically separate zones within an Azure region. Each Availability Zone has a distinct power source, network, and cooling. Deploying VMs across Availability Zones helps to protect an application against datacenter-wide failures. Not all regions support Availability Zones. For a list of supported regions and services, see [What are Availability Zones in Azure?](#).

If you plan to use Availability Zones in your deployment, first validate that your application architecture and codebase support this configuration. If you deploy commercial software, consult with the software vendor and test adequately before deploying into production. An application must maintain state and prevent loss of data during an outage within the configured zone. The application must support running in an elastic and distributed infrastructure with no hard-coded infrastructure components.

- **Azure Site Recovery** replicates Azure Virtual Machines to another Azure region for business continuity (BC) and disaster recovery (DR) needs. You can conduct periodic DR drills to ensure that you meet the compliance needs. The VM is replicated with the specified settings to the selected region so you can recover your applications in the event of outages in the source region. For more information, see [Set up disaster recovery to a secondary Azure region for an Azure VM](#).

During testing, verify that the *recovery time objective* (RTO) and *recovery point objective* (RPO) meet your needs. RTO is the maximum time an application is unavailable after an incident, and RPO is the maximum duration of data loss during a disaster.

- **Paired regions** are created using Azure Traffic Manager to distribute Internet traffic to different regions, protecting an application against a regional outage. Each Azure region is paired with another region. Together, these regions form a [regional pair](#). To meet data residency requirements for tax and law enforcement jurisdiction purposes, regional pairs are located within the same geography (with the exception of Brazil South).

To improve application resiliency, Azure serializes platform updates (planned maintenance) across each region pair, so only one paired region is updated at a time.

- When you design a multiregion application, take into account that network latency across regions is higher than within a region. For example, if you replicate a database to enable failover, use synchronous data replication within a region but asynchronous data replication across regions.

The following table compares the redundancy factors across several resiliency strategies:

Availability set	Availability Zone	Azure Site Recovery/Paired region
------------------	-------------------	-----------------------------------

	Availability set	Availability Zone	Azure Site Recovery/Paired region
Scope of failure	Rack	Datacenter	Region
Request routing	Azure Load Balancer	Cross-zone Load Balancer	Azure Traffic Manager
Network latency	Very low	Low	Mid to high
Virtual network	Azure Virtual Network	Azure Virtual Network	Cross-region Virtual Network peering

Complete Azure redundancy tasks

Use the following tasks to meet redundancy requirements:

- **Deploy multiple instances of services.** If your application depends on a single instance of a service, it creates a single point of failure. Provisioning multiple instances improves both resiliency and scalability. For [Azure App Service](#), select an [App Service plan](#) that offers multiple instances. For [Azure Virtual Machines](#), ensure that your architecture has more than one VM and that each VM is included in an [availability set](#).
- **Replicate VMs using Azure Site Recovery.** When you replicate Azure VMs using [Site Recovery](#), all the VM disks are continuously replicated to the target region asynchronously. The recovery points are created every few minutes, giving an RPO on the order of minutes.
- **Consider deploying your application across multiple regions.** If your application is deployed to a single region, and the region becomes unavailable, your application will also be unavailable. This may be unacceptable under the terms of your application's SLA. If so, consider deploying your application and its services across multiple regions. A multiregion deployment can use an *active-active* or *active-passive* configuration. An active-active configuration distributes requests across multiple active regions. An active-passive configuration keeps warm instances in the secondary region, but doesn't send traffic there unless the primary region fails. For multiregion deployments, we recommend deploying to paired regions, described above. For more information, see [Business continuity and disaster recovery \(BCDR\): Azure Paired Regions](#).
- **Use Azure Traffic Manager to route your application's traffic to different regions.** [Azure Traffic Manager](#) performs load-balancing at the DNS level and routes traffic to different regions based on the [traffic routing](#) method and the health of your application's endpoints. Without Traffic Manager, you are limited to a single region for your deployment, which constrains scale, increases latency for some users, and causes application downtime, in the case of a region-wide service disruption.
- **Configure Azure Application Gateway to use multiple instances.** Depending on your application's requirements, an [Azure Application Gateway](#) may be better suited to distributing requests to your application's services. However, single instances of the Application Gateway service are not guaranteed by an SLA, so it's possible that your application could fail if the Application Gateway instance fails. Provision more than one medium or larger instance to guarantee availability of the service under the terms of the [Application Gateway SLA](#).

Design for scalability

Scalability is the ability of a system to handle increased load and is one of the [pillars of software quality](#). Scalability tasks during the architecting phase include:

- **Partition workloads.** Design parts of the process to be discrete and decomposable. Minimize the size of each part. This allows the component parts to be distributed in a way that maximizes use of each compute unit. It also makes it easier to scale the application by adding instances of specific resources. For complex domains, consider adopting a [microservices architecture](#).

- **Design for scaling.** Scaling allows applications to react to variable load by increasing and decreasing the number of instances of roles, queues, and other services. However, the application must be designed with this in mind. For example, the application and the services it uses must be stateless to allow requests to be routed to any instance. Having stateless services also means that adding or removing an instance does not adversely impact current users.
- **Plan for growth with scale units.** For each resource, know the upper scaling limits, and use sharding or decomposition to go beyond those limits. Design the application so that it's easily scaled by adding one or more scale units. Determine the scale units for the system in terms of well-defined sets of resources. This makes applying scale-out operations easier and less prone to negative impact caused by a lack of resources in some part of the overall system. For example, adding X number of front-end VMs might require Y number of additional queues and Z number of storage accounts to handle the additional workload. So a scale unit could consist of X VM instances, Y queues, and Z storage accounts.
- **Avoid client affinity.** Where possible, ensure that the application doesn't require affinity. Requests can then be routed to any instance, and the number of instances is irrelevant. This also avoids the overhead of storing, retrieving, and maintaining state information for each user.
- **Take advantage of platform autoscaling features.** Use built-in autoscaling features when possible, rather than custom or third-party mechanisms. Use scheduled scaling rules, where possible, to ensure that resources are available without a startup delay, but add reactive autoscaling to the rules, where appropriate, to cope with unexpected changes in demand. For more information, see [Autoscaling guidance](#).

If your application isn't configured to scale out automatically as load increases, it's possible that your application's services will fail if they become saturated with user requests. For more information, see the following articles:

- General: [Scalability checklist](#)
 - Azure App Service: [Scale instance count manually or automatically](#)
 - Cloud Services: [How to autoscale a Cloud Service](#)
 - Virtual machines: [Automatic scaling and virtual machine scale sets](#)
- **Offload intensive CPU/IO tasks as background tasks.** If a request to a service is expected to take a long time to run or may absorb considerable resources, offload the processing to a separate task. Use background jobs to execute these tasks. This strategy enables the service to continue receiving further requests and to remain responsive. For more information, see [Background jobs guidance](#).
 - **Distribute the workload for background tasks.** If there are many background tasks or if the tasks require considerable time or resources, spread the work across multiple compute units. For one possible solution, see the [Competing Consumers pattern](#).
 - **Consider moving toward a *shared-nothing* architecture.** This architecture uses independent, self-sufficient nodes that have no single point of contention (such as shared services or storage). In theory, such a system can scale almost indefinitely. Although a fully shared-nothing approach is usually not practical, it may provide opportunities to design for better scalability. Good examples of moving toward a shared-nothing architecture include partitioning data and avoiding the use of server-side session state and client affinity.
 - **Design your application's storage requirements to fall within Azure Storage scalability and performance targets.** Azure Storage is designed to function within predefined scalability and performance targets, so design your application to use storage within those targets. If you exceed these targets, your application will experience storage throttling. To avoid throttling, provision additional storage accounts. If you run up against the storage account limit, provision additional Azure subscriptions and then provision additional storage accounts there. For more information, see [Azure Storage scalability and performance targets](#).
 - **Select the right VM size for your application.** Measure the actual CPU, memory, disk, and I/O of your VMs in production, and verify that the VM size you've selected is sufficient. If not, your application may experience capacity issues as the VMs approach their limits. VM sizes are described in detail in [Sizes for virtual machines in Azure](#).

Determine subscription and service requirements

Choose the right subscription and service features for your app by working through these tasks:

- **Evaluate requirements against [Azure subscription and service limits](#).** *Azure subscriptions* have limits on certain resource types, such as number of resource groups, cores, and storage accounts. If your application requirements exceed Azure subscription limits, create another Azure subscription and provision sufficient resources there. Individual Azure services have consumption limits — for example, limits on storage, throughput, number of connections, requests per second, and other metrics. Your application will fail if it attempts to use resources beyond these limits, resulting in service throttling and possible downtime for affected users. Depending on the specific service and your application requirements, you can often avoid these limits by scaling up (for example, choosing another pricing tier) or scaling out (such as adding new instances).
- **Determine how many storage accounts you need.** Azure allows a specific number of storage accounts per subscription. For more information, see [Azure subscription and service limits, quotas, and constraints](#).
- **Select the right service tier for Azure SQL Database.** If your application uses Azure SQL Database, select the appropriate service tier. If the tier cannot handle your application's database transaction unit (DTU) requirements, your data use will be throttled. For more information on selecting the correct service plan, see [SQL Database options and performance: Understand what's available in each service tier](#).
- **Provision sufficient request units (RUs) in Azure Cosmos DB.** With Azure Cosmos DB, you pay for the throughput you provision and the storage you consume on an hourly basis. The cost of all database operations is normalized as RUs, which abstracts the system resources such as CPU, IOPS, and memory. For more information, see [Request Units in Azure Cosmos DB](#).

Load-balance as needed

Proper load-balancing allows you to meet availability requirements and to minimize costs associated with availability.

- **Use load-balancing to distribute requests.** Load-balancing distributes your application's requests to healthy service instances by removing unhealthy instances from rotation. If your service uses Azure App Service or Azure Cloud Services, it's already load-balanced for you. However, if your application uses Azure VMs, you need to provision a load-balancer. For more information, see [What is Azure Load Balancer?](#)

You can use Azure Load Balancer to:

- Load-balance incoming Internet traffic to your VMs. This configuration is known as a [public Load Balancer](#).
 - Load-balance traffic across VMs inside a virtual network. You can also reach a Load Balancer front end from an on-premises network in a hybrid scenario. Both scenarios use a configuration that is known as an [internal Load Balancer](#).
 - Port forward traffic to an itemized port on specific VMs with inbound network address translation (NAT) rules.
 - Provide [outbound connectivity](#) for VMs inside your virtual network by using a public Load Balancer.
- **Balance loads across regions with a traffic manager, such as Azure Traffic Manager.** To load-balance traffic across regions requires a traffic management solution, and Azure provides [Traffic Manager](#). You can also take advantage of third-party services that provide similar traffic-management capabilities.

Implement resiliency strategies

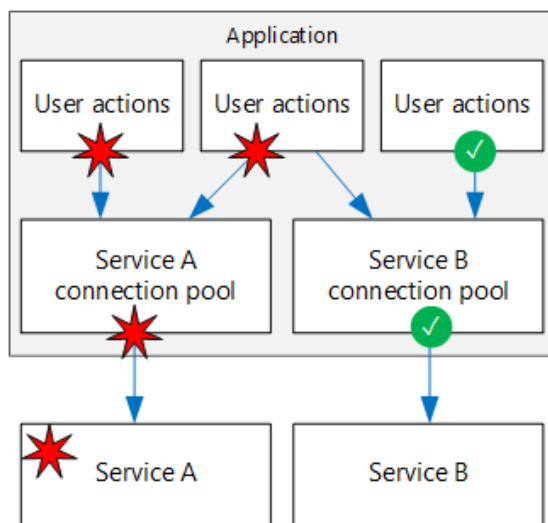
This section describes some common resiliency strategies. Most of these strategies are not limited to a particular technology. The descriptions summarize the general idea behind each technique and include links to further reading.

- **Implement resiliency patterns** for remote operations, where appropriate. If your application depends on communication between remote services, follow [design patterns](#) for dealing with transient failures.
- **Retry transient failures.** These can be caused by momentary loss of network connectivity, a dropped database connection, or a timeout when a service is busy. Often, a transient failure can be resolved by retrying the request.

- For many Azure services, the client software development kit (SDK) implements automatic retries in a way that is transparent to the caller. See [Retry guidance for specific services](#).
- Or implement the [Retry pattern](#) to help the application handle anticipated, temporary failures transparently when it tries to connect to a service or network resource.
- **Use a circuit breaker** to handle faults that might take a variable amount of time to fix. The [Circuit Breaker pattern](#) can prevent an application from repeatedly trying an operation that is likely to fail. The circuit breaker wraps calls to a service and tracks the number of recent failures. If the failure count exceeds a threshold, the circuit breaker starts returning an error code without calling the service. This gives the service time to recover and helps avoid cascading failures.
- **Isolate critical resources.** Failures in one subsystem can sometimes cascade, resulting in failures in other parts of the application. This can happen if a failure prevents resources such as threads or sockets from being freed, leading to resource exhaustion. To avoid this, you can partition a system into isolated groups so that a failure in one partition does not bring down the entire system.

Here are some examples of this technique, which is sometimes called the [Bulkhead pattern](#):

- Partition a database (for example, by tenant), and assign a separate pool of web server instances for each partition.
- Use separate thread pools to isolate calls to different services. This helps to prevent cascading failures if one of the services fails. For an example, see the Netflix [Hystrix library](#).
- Use [containers](#) to limit the resources available to a particular subsystem.



- **Apply [compensating transactions](#).** A compensating transaction is a transaction that undoes the effects of another completed transaction. In a distributed system, it can be difficult to achieve strong transactional consistency. Compensating transactions help to achieve consistency by using a series of smaller, individual transactions that can be undone at each step. For example, to book a trip, a customer might reserve a car, a hotel room, and a flight. If one of these steps fails, the entire operation fails. Instead of trying to use a single distributed transaction for the entire operation, you can define a compensating transaction for each step.
- **Implement asynchronous operations, whenever possible.** Synchronous operations can monopolize resources and block other operations while the caller waits for the process to complete. Design each part of your application to allow for asynchronous operations, whenever possible. For more information on how to implement asynchronous programming in C#, see [Asynchronous Programming](#).

Ensure that availability meets SLAs

Availability is the proportion of time that a system is functional and working, and it is one of the [pillars of software quality](#). Use the tasks in this section to review your application architecture from an availability standpoint to make sure

that your availability meets your SLAs.

- **Avoid any single point of failure.** All components, services, resources, and compute instances should be deployed as multiple instances to prevent a single point of failure from affecting availability. Authentication mechanisms can also be a single point of failure. Design the application to be configurable to use multiple instances and to automatically detect failures and redirect requests to non-failed instances, if the platform doesn't do this automatically.
- **Decompose workloads by service-level objective.** If a service is composed of critical and less-critical workloads, manage them differently and specify the service features and number of instances to meet their availability requirements.
- **Minimize and understand service dependencies.** Minimize the number of different services used, where possible. Ensure that you understand all the feature and service dependencies that exist in the system. In particular, understand the overall impact of failure or reduced performance in each dependency.
- **Design tasks and messages to be *idempotent*, where possible.** An operation is idempotent if it can be repeated multiple times and produce the same result. This can ensure that duplicated requests don't cause problems. Message consumers and the operations they carry out should be idempotent so that repeating a previously executed operation does not render the results invalid. This may mean detecting duplicated messages or ensuring consistency by using an optimistic approach to handling conflicts.
- **Configure request timeouts.** Services and resources may become unavailable, causing requests to fail. Ensure that the timeouts you apply are appropriate for each service or resource and for the client that is accessing them. In some cases, you might allow a longer timeout for a particular instance of a client, depending on the context and other actions that the client is performing. Short timeouts may cause excessive retry operations for services and resources that have considerable latency. Long timeouts can cause blocking, if a large number of requests are queued, waiting for a service or resource to respond.
- **Use a message broker that implements high availability for critical transactions.** Many cloud applications use messaging to trigger asynchronous tasks. To guarantee delivery of messages, the messaging system should provide high availability. [Azure Service Bus messaging](#) implements *at least once* semantics, which means that a message is guaranteed to be delivered at least once. Duplicate messages may be delivered under certain circumstances. If message processing is idempotent (see the previous item), repeated delivery should not be a problem.
- **Throttle high-volume users.** Sometimes, a small number of users creates excessive load. This can have an impact on other users and can reduce the overall availability of your application. When a single client makes an excessive number of requests, the application might throttle the client for a certain period. During the throttling period, the application refuses some or all of the requests from that client. The threshold for throttling often depends on the customer's service tier. For more information, see [Throttling.pattern](#).

Throttling does not imply that the client was necessarily acting maliciously — only that it exceeded its service quota. In some cases, a consumer might consistently exceed their quota or otherwise behave badly. In that case, you might go further and block the user. Typically, this is done by blocking an API key or an IP address range.

- **Design applications to gracefully degrade.** The load on an application may exceed the capacity of one or more parts, causing reduced availability and failed connections. Scaling can mitigate this problem, but it may reach a limit imposed by other factors, such as resource availability or cost. When an application reaches a resource limit, it should take appropriate action to minimize the impact for the user. For example, in an e-commerce system, if the order-processing subsystem is under strain or fails, it can be temporarily disabled while allowing other functionality, such as browsing the product catalog. It might be appropriate to postpone requests to a failing subsystem — for example, still enabling customers to submit orders but saving them for later processing, when the orders subsystem is available again.

- **Gracefully handle rapid burst events.** Most applications need to handle varying workloads over time. Autoscaling can help to handle the load, but it may take some time for additional instances to come online and handle requests. To prevent bursts of activity from overwhelming the application, design it to queue requests to the services it uses and to degrade gracefully when queues are near capacity. Ensure there is sufficient performance and capacity available under non-burst conditions to drain the queues and handle outstanding requests. For more information, see the [Queue-Based Load Leveling pattern](#).
- **Compose or fail back to multiple components.** Design applications to use multiple instances without affecting operation and existing connections, where possible. To maximize availability, use multiple instances and distribute requests between them, and detect and avoid sending requests to failed instances.
- **Fail back to a different service or workflow.** For example, if writing to SQL Database fails, temporarily store data in Blob storage or Redis Cache. Provide a way to replay the writes to SQL Database when the service becomes available. In some cases, a failed operation may have an alternative action that allows the application to continue to work, even when a component or service fails. If possible, detect failures and redirect requests to other services while the primary service is offline.
- **Use load leveling to smooth out spikes in traffic.** Applications may experience sudden spikes in traffic, which can overwhelm services on the back end. If a back-end service cannot respond to requests quickly enough, the pending requests may accumulate or the service may throttle the application. To avoid this, you can use a queue as a buffer. When there is a new work item, instead of calling the back-end service immediately, the application queues a work item to run asynchronously. The queue acts as a buffer that smooths out peaks in the load. For more information, see [Queue-Based Load Leveling pattern](#).

Manage your data

How you manage your data plays directly into the availability of your application. The tasks in this section can help you create a management plan to help ensure availability.

- **Replicate data and understand the replication methods for your application's data stores.** Replicating data is a general strategy for handling non-transient failures in a data store. Consider both the read and write paths. Depending on the storage technology, you might have multiple writable replicas or you might have a single writable replica and multiple read-only replicas. To maximize availability, replicas can be placed in multiple regions. However, this approach increases the latency when replicating the data. Typically, replicating across regions is done asynchronously, which implies an eventual consistency model and potential data loss if a replica fails.

You can use [Azure Site Recovery](#) to replicate Azure Virtual Machines from one region to another. Site Recovery replicates data continuously to the target region. When an outage occurs at your primary site, you fail over to a secondary location.

- **Ensure that no single user account has access to both production and backup data.** Your data backups are compromised if one single user account has permission to write to both production and backup sources. A malicious user could purposely delete all your data, and a regular user could accidentally delete it. Design your application to limit the permissions of each user account. Only grant write access to users who require it, and grant access to either production or backup, but not both.
- **Document and test your data store failover and fallback process.** If a data store fails catastrophically, a human operator must follow a set of documented instructions to fail over to a new data store. If the documented steps have errors, an operator won't be able to successfully follow them and to fail over the resource. Regularly test the instruction steps to verify that an operator who follows the documentation can successfully fail over and fail back.
- **Back up your data and validate your data backups.** Regularly run a script to validate data integrity, schema, and queries to ensure that backup data is what you expect. Log and report any inconsistencies so the backup service can be repaired.

- **Use periodic backup and point-in-time restore.** Regularly and automatically back up data that is not preserved elsewhere. Verify that you can reliably restore both the data and the application itself if failure occurs. Ensure that backups meet your RPO. Data replication isn't a backup feature, because human error or malicious operations can corrupt data across all the replicas. The backup process must be secure to protect the data in transit and in storage. Databases can usually be recovered to a previous point in time by using transaction logs. For more information, see [Recover from data corruption or accidental deletion](#).

- **Consider using a geo-redundant storage account.** Data stored in an Azure Storage account is always replicated locally. However, there are multiple replication strategies to choose from when a storage account is provisioned. To protect your application data against the rare case when an entire region becomes unavailable, select [Azure Read-Access Geo-Redundant Storage \(RA-GRS\)](#).

ⓘ **Note**

For VMs, do not rely on RA-GRS replication to restore the VM disks (VHD files). Instead, use [Azure Backup](#).

- **Consider deploying reference data to multiple regions.** Reference data is read-only data that supports application functionality. It typically doesn't change often. Although restoring from backup is one way to handle region-wide service disruptions, the RTO is relatively long. When you deploy the application to a secondary region, some strategies can improve the RTO for reference data.

Because reference data changes infrequently, you can improve the RTO by maintaining a permanent copy in the secondary region. This eliminates the time required to restore backups after a disaster. To meet the multiple-region disaster recovery requirements, you must deploy the application and the reference data together in multiple regions.

- **Use optimistic concurrency and eventual consistency.** Transactions that block access to resources through locking (*pessimistic concurrency*) can cause poor performance and reduce availability. These problems can become especially acute in distributed systems. In many cases, careful design and techniques, such as partitioning, can minimize the chances of conflicting updates occurring. If data is replicated or read from a separately updated store, the data will only be eventually consistent. But the advantages usually outweigh the impact on availability of using transactions to ensure immediate consistency.
- **Use active geo-replication for SQL Database to replicate changes to a secondary database.** Active geo-replication for SQL Database automatically replicates database changes to secondary databases in the same region or a different region. For more information, see [Creating and using active geo-replication](#).

Alternatively, you can take a more manual approach by using the **DATABASE COPY** command to create a backup copy of the database with transactional consistency. You can also use the import/export service of Azure SQL Database, which supports exporting databases to BACPAC files (compressed files containing your database schema and associated data) that are stored in Azure Blob storage. Azure Storage creates two replicas of the backup file in the same region. However, the frequency of the backup process determines your RPO, which is the amount of data you might lose in disaster scenarios. For example, if you back up data every hour, and a disaster occurs two minutes before the backup, you will lose 58 minutes of data. Also, to protect against a region-wide service disruption, you should copy the BACPAC files to an alternate region. For more information, see [Overview of business continuity with Azure SQL Database](#).

- **Use geo-backups for SQL Data Warehouse.** For SQL Data Warehouse, use [geo-backups](#) to restore to a paired region for disaster recovery. These backups are taken every 24 hours and can be restored within 20 minutes in the paired region. This feature is on by default for all SQL Data Warehouse instances. For more information on how to restore your data warehouse, see [Restore from an Azure geographical region using PowerShell](#).
- **Replicate VM disks using Azure Site Recovery.** When you replicate Azure VMs using [Site Recovery](#), all the VM disks are continuously replicated to the target region asynchronously. The recovery points are created every few minutes. This gives you an RPO on the order of minutes.

- **Back up SQL Server running on VMs or configure a log-shipping session.** For SQL Server running on VMs, there are two options: traditional backups and log shipping. Traditional backups enable you to restore to a specific point in time, but the recovery process is slow. Restoring traditional backups requires that you start with an initial full backup and then apply any backups taken after that. The second option is to configure a log-shipping session to delay the restore of log backups (for example, by two hours). This provides a window to recover from errors made on the primary.
- **Use a custom process or third-party tool for Azure Storage backup.** For Azure Storage, you can develop a custom backup process or use a third-party backup tool. Most application designs have additional complexities, in which storage resources reference each other. For example, consider a SQL database with a column that links to a blob in Azure Storage. If the backups do not happen simultaneously, the database might have a pointer to a blob that was not backed up before the failure. The application or disaster recovery plan must implement processes to handle this inconsistency after a recovery.
- **Use the native replication or snapshot capabilities for other data platforms hosted on VMs.** Other data platforms, such as Elasticsearch or MongoDB, have their own capabilities and considerations when creating an integrated backup and restore process. For these data platforms, the general recommendation is to use any native or available integration-based replication or snapshot capabilities. If those capabilities don't exist or aren't suitable, consider using Azure Backup or disk snapshots to create a point-in-time copy of application data. In all cases, it's important to determine how to achieve consistent backups, especially when application data spans multiple files systems or multiple drives are combined into a single file system.
- **Understand the replication methods for your application's data sources.** Your application data will be stored in different data sources and will have varied availability requirements. Evaluate the replication methods for each type of data storage in Azure, including [Azure Storage redundancy](#) and [SQL Database active geo-replication](#) to ensure that your application's data requirements are satisfied. If you replicate Azure VMs using [Site Recovery](#), all the VM disks are continuously replicated to the target region asynchronously. The recovery points are created every few minutes.
- **Establish data strategies for disaster recovery.** Proper data handling is a challenging aspect of any disaster recovery plan. During the recovery process, data restoration typically takes the most time. Different choices for reducing functionality result in difficult challenges for data recovery and consistency.

Next steps

Test for resiliency and availability