# Failure mode analysis for Azure applications

05/07/2018 • 17 minutes to read • Contributors all

**In this article**



Failure mode analysis (FMA) is a process for building resiliency into a system, by identifying possible failure points in the system. The FMA should be part of the architecture and design phases, so that you can build failure recovery into the system from the beginning.

Here is the general process to conduct an FMA:

1. Identify all of the components in the system. Include external dependencies, such as identity providers, third-party services, and so on.

2. For each component, identify potential failures that could occur. A single component may have more than one failure mode. For example, you should consider read failures and write failures separately, because the impact and possible mitigations will be different.

3. Rate each failure mode according to its overall risk. Consider these factors:

    - What is the likelihood of the failure. Is it relatively common? Extremely rare? You don't need exact numbers; the purpose is to help rank the priority.
    - What is the impact on the application, in terms of availability, data loss, monetary cost, and business disruption?

4. For each failure mode, determine how the application will respond and recover. Consider tradeoffs in cost and application complexity.

As a starting point for your FMA process, this article contains a catalog of potential failure modes and their mitigations. The catalog is organized by technology or Azure service, plus a general category for application-level design. The catalog is not exhaustive, but covers many of the core Azure services.

## App Service

### App Service app shuts down.

**Detection**. Possible causes:

- Expected shutdown
  - An operator shuts down the application; for example, using the Azure portal.
  - The app was unloaded because it was idle. (Only if the `Always On` setting is disabled.)
- Unexpected shutdown
  - The app crashes.
  - An App Service VM instance becomes unavailable.

Application_End logging will catch the app domain shutdown (soft process crash) and is the only way to catch the application domain shutdowns.

**Recovery:**

- If the shutdown was expected, use the application's shutdown event to shut down gracefully. For example, in ASP.NET, use the `Application_End` method.
- If the application was unloaded while idle, it is automatically restarted on the next request. However, you will incur the "cold start" cost.
- To prevent the application from being unloaded while idle, enable the `Always On` setting in the web app. See Configure web apps in Azure App Service.
- To prevent an operator from shutting down the app, set a resource lock with `ReadOnly` level. See Lock resources with Azure Resource Manager.
- If the app crashes or an App Service VM becomes unavailable, App Service automatically restarts the app.

**Diagnostics**. Application logs and web server logs. See Enable diagnostics logging for web apps in Azure App Service.

### A particular user repeatedly makes bad requests or overloads the system.

**Detection**. Authenticate users and include user ID in application logs.

**Recovery:**

- Use Azure API Management to throttle requests from the user. See Advanced request throttling with Azure API Management
- Block the user.

**Diagnostics**. Log all authentication requests.

### A bad update was deployed.

**Detection**. Monitor the application health through the Azure Portal (see Monitor Azure web app performance) or implement the health endpoint monitoring pattern.

**Recovery:**. Use multiple deployment slots and roll back to the last-known-good deployment. For more information, see Basic web application.

## Azure Active Directory

### OpenID Connect (OIDC) authentication fails.

**Detection**. Possible failure modes include:

1. Azure AD is not available, or cannot be reached due to a network problem. Redirection to the authentication endpoint fails, and the OIDC middleware throws an exception.
2. Azure AD tenant does not exist. Redirection to the authentication endpoint returns an HTTP error code, and the OIDC middleware throws an exception.
3. User cannot authenticate. No detection strategy is necessary; Azure AD handles login failures.

**Recovery:**

1. Catch unhandled exceptions from the middleware.
2. Handle `AuthenticationFailed` events.
3. Redirect the user to an error page.
4. User retries.

# Azure Search

### Writing data to Azure Search fails.

**Detection**. Catch `Microsoft.Rest.Azure.CloudException` errors.

**Recovery:**

The Search .NET SDK automatically retries after transient failures. Any exceptions thrown by the client SDK should be treated as non-transient errors.

The default retry policy uses exponential back-off. To use a different retry policy, call `SetRetryPolicy` on the `SearchIndexClient` or `SearchServiceClient` class. For more information, see Automatic Retries.

**Diagnostics**. Use Search Traffic Analytics.

### Reading data from Azure Search fails.

**Detection**. Catch `Microsoft.Rest.Azure.CloudException` errors.

**Recovery:**

The Search .NET SDK automatically retries after transient failures. Any exceptions thrown by the client SDK should be treated as non-transient errors.

The default retry policy uses exponential back-off. To use a different retry policy, call `SetRetryPolicy` on the `SearchIndexClient` or `SearchServiceClient` class. For more information, see Automatic Retries.

**Diagnostics**. Use Search Traffic Analytics.

# Cassandra

### Reading or writing to a node fails.

**Detection**. Catch the exception. For .NET clients, this will typically be `System.Web.HttpException`. Other client may have other exception types. For more information, see Cassandra error handling done right.

**Recovery:**

- Each Cassandra client has its own retry policies and capabilities. For more information, see Cassandra error handling done right.
- Use a rack-aware deployment, with data nodes distributed across the fault domains.

- Deploy to multiple regions with local quorum consistency. If a non-transient failure occurs, fail over to another region.

**Diagnostics**. Application logs

# Cloud Service

### Web or worker roles are unexpectedly being shut down.

**Detection**. The RoleEnvironment.Stopping event is fired.

**Recovery**. Override the RoleEntryPoint.OnStop method to gracefully clean up. For more information, see The Right Way to Handle Azure OnStop Events (blog).

# Cosmos DB

### Reading data fails.

**Detection**. Catch `System.Net.Http.HttpRequestException` or `Microsoft.Azure.Documents.DocumentClientException`.

**Recovery:**

- The SDK automatically retries failed attempts. To set the number of retries and the maximum wait time, configure `ConnectionPolicy.RetryOptions`. Exceptions that the client raises are either beyond the retry policy or are not transient errors.
- If Cosmos DB throttles the client, it returns an HTTP 429 error. Check the status code in the `DocumentClientException`. If you are getting error 429 consistently, consider increasing the throughput value of the collection.
  - If you are using the MongoDB API, the service returns error code 16500 when throttling.
- Replicate the Cosmos DB database across two or more regions. All replicas are readable. Using the client SDKs, specify the `PreferredLocations` parameter. This is an ordered list of Azure regions. All reads will be sent to the first available region in the list. If the request fails, the client will try the other regions in the list, in order. For more information, see How to set up Azure Cosmos DB global distribution using the SQL API.

**Diagnostics**. Log all errors on the client side.

### Writing data fails.

**Detection**. Catch `System.Net.Http.HttpRequestException` or `Microsoft.Azure.Documents.DocumentClientException`.

**Recovery:**

- The SDK automatically retries failed attempts. To set the number of retries and the maximum wait time, configure `ConnectionPolicy.RetryOptions`. Exceptions that the client raises are either beyond the retry policy or are not transient errors.
- If Cosmos DB throttles the client, it returns an HTTP 429 error. Check the status code in the `DocumentClientException`. If you are getting error 429 consistently, consider increasing the throughput value of the collection.
- Replicate the Cosmos DB database across two or more regions. If the primary region fails, another region will be promoted to write. You can also trigger a failover manually. The SDK does automatic discovery and routing, so application code continues to work after a failover. During the failover period (typically minutes), write operations

will have higher latency, as the SDK finds the new write region. For more information, see How to set up Azure Cosmos DB global distribution using the SQL API.
- As a fallback, persist the document to a backup queue, and process the queue later.

**Diagnostics**. Log all errors on the client side.

# Elasticsearch

### Reading data from Elasticsearch fails.

**Detection**. Catch the appropriate exception for the particular Elasticsearch client being used.

**Recovery:**

- Use a retry mechanism. Each client has its own retry policies.
- Deploy multiple Elasticsearch nodes and use replication for high availability.

For more information, see Running Elasticsearch on Azure.

**Diagnostics**. You can use monitoring tools for Elasticsearch, or log all errors on the client side with the payload. See the 'Monitoring' section in Running Elasticsearch on Azure.

### Writing data to Elasticsearch fails.

**Detection**. Catch the appropriate exception for the particular Elasticsearch client being used.

**Recovery:**

- Use a retry mechanism. Each client has its own retry policies.
- If the application can tolerate a reduced consistency level, consider writing with `write_consistency` setting of `quorum`.

For more information, see Running Elasticsearch on Azure.

**Diagnostics**. You can use monitoring tools for Elasticsearch, or log all errors on the client side with the payload. See the 'Monitoring' section in Running Elasticsearch on Azure.

# Queue storage

### Writing a message to Azure Queue storage fails consistently.

**Detection**. After *N* retry attempts, the write operation still fails.

**Recovery:**

- Store the data in a local cache, and forward the writes to storage later, when the service becomes available.
- Create a secondary queue, and write to that queue if the primary queue is unavailable.

**Diagnostics**. Use storage metrics.

### The application cannot process a particular message from the queue.

**Detection**. Application specific. For example, the message contains invalid data, or the business logic fails for some reason.

**Recovery:**

Move the message to a separate queue. Run a separate process to examine the messages in that queue.

Consider using Azure Service Bus Messaging queues, which provides a dead-letter queue functionality for this purpose.

> **Note**
>
> If you are using Storage queues with WebJobs, the WebJobs SDK provides built-in poison message handling. See How to use Azure queue storage with the WebJobs SDK.

**Diagnostics**. Use application logging.

# Redis Cache

### Reading from the cache fails.

**Detection**. Catch `StackExchange.Redis.RedisConnectionException`.

**Recovery:**

1. Retry on transient failures. Azure Redis cache supports built-in retry through See Redis Cache retry guidelines.
2. Treat non-transient failures as a cache miss, and fall back to the original data source.

**Diagnostics**. Use Redis Cache diagnostics.

### Writing to the cache fails.

**Detection**. Catch `StackExchange.Redis.RedisConnectionException`.

**Recovery:**

1. Retry on transient failures. Azure Redis cache supports built-in retry through See Redis Cache retry guidelines.
2. If the error is non-transient, ignore it and let other transactions write to the cache later.

**Diagnostics**. Use Redis Cache diagnostics.

# SQL Database

### Cannot connect to the database in the primary region.

**Detection**. Connection fails.

**Recovery:**

Prerequisite: The database must be configured for active geo-replication. See SQL Database Active Geo-Replication.

- For queries, read from a secondary replica.
- For inserts and updates, manually fail over to a secondary replica. See Initiate a planned or unplanned failover for Azure SQL Database.

The replica uses a different connection string, so you will need to update the connection string in your application.

### Client runs out of connections in the connection pool.

**Detection**. Catch `System.InvalidOperationException` errors.

**Recovery:**

- Retry the operation.
- As a mitigation plan, isolate the connection pools for each use case, so that one use case can't dominate all the connections.
- Increase the maximum connection pools.

**Diagnostics**. Application logs.

### Database connection limit is reached.

**Detection**. Azure SQL Database limits the number of concurrent workers, logins, and sessions. The limits depend on the service tier. For more information, see Azure SQL Database resource limits.

To detect these errors, catch `System.Data.SqlClient.SqlException` and check the value of `SqlException.Number` for the SQL error code. For a list of relevant error codes, see SQL error codes for SQL Database client applications: Database connection error and other issues.

**Recovery**. These errors are considered transient, so retrying may resolve the issue. If you consistently hit these errors, consider scaling the database.

**Diagnostics**. - The sys.event_log query returns successful database connections, connection failures, and deadlocks.

- Create an alert rule for failed connections.
- Enable SQL Database auditing and check for failed logins.

## Service Bus Messaging

### Reading a message from a Service Bus queue fails.

**Detection**. Catch exceptions from the client SDK. The base class for Service Bus exceptions is MessagingException. If the error is transient, the `IsTransient` property is true.

For more information, see Service Bus messaging exceptions.

**Recovery:**

1. Retry on transient failures. See Service Bus retry guidelines.
2. Messages that cannot be delivered to any receiver are placed in a *dead-letter queue*. Use this queue to see which messages could not be received. There is no automatic cleanup of the dead-letter queue. Messages remain there until you explicitly retrieve them. See Overview of Service Bus dead-letter queues.

### Writing a message to a Service Bus queue fails.

**Detection**. Catch exceptions from the client SDK. The base class for Service Bus exceptions is MessagingException. If the error is transient, the `IsTransient` property is true.

For more information, see Service Bus messaging exceptions.

**Recovery:**

1. The Service Bus client automatically retries after transient errors. By default, it uses exponential back-off. After the maximum retry count or maximum timeout period, the client throws an exception. For more information, see Service Bus retry guidelines.

2. If the queue quota is exceeded, the client throws QuotaExceededException. The exception message gives more details. Drain some messages from the queue before retrying, and consider using the Circuit Breaker pattern to

avoid continued retries while the quota is exceeded. Also, make sure the BrokeredMessage.TimeToLive property is not set too high.

3. Within a region, resiliency can be improved by using partitioned queues or topics. A non-partitioned queue or topic is assigned to one messaging store. If this messaging store is unavailable, all operations on that queue or topic will fail. A partitioned queue or topic is partitioned across multiple messaging stores.

4. For additional resiliency, create two Service Bus namespaces in different regions, and replicate the messages. You can use either active replication or passive replication.

   - Active replication: The client sends every message to both queues. The receiver listens on both queues. Tag messages with a unique identifier, so the client can discard duplicate messages.
   - Passive replication: The client sends the message to one queue. If there is an error, the client falls back to the other queue. The receiver listens on both queues. This approach reduces the number of duplicate messages that are sent. However, the receiver must still handle duplicate messages.

   For more information, see GeoReplication sample and Best practices for insulating applications against Service Bus outages and disasters.

### Duplicate message.

**Detection**. Examine the `MessageId` and `DeliveryCount` properties of the message.

**Recovery:**

- If possible, design your message processing operations to be idempotent. Otherwise, store message IDs of messages that are already processed, and check the ID before processing a message.
- Enable duplicate detection, by creating the queue with `RequiresDuplicateDetection` set to true. With this setting, Service Bus automatically deletes any message that is sent with the same `MessageId` as a previous message. Note the following:
  - This setting prevents duplicate messages from being put into the queue. It doesn't prevent a receiver from processing the same message more than once.
  - Duplicate detection has a time window. If a duplicate is sent beyond this window, it won't be detected.

**Diagnostics**. Log duplicated messages.

### The application can't process a particular message from the queue.

**Detection**. Application specific. For example, the message contains invalid data, or the business logic fails for some reason.

**Recovery:**

There are two failure modes to consider.

- The receiver detects the failure. In this case, move the message to the dead-letter queue. Later, run a separate process to examine the messages in the dead-letter queue.
- The receiver fails in the middle of processing the message — for example, due to an unhandled exception. To handle this case, use `PeekLock` mode. In this mode, if the lock expires, the message becomes available to other receivers. If the message exceeds the maximum delivery count or the time-to-live, the message is automatically moved to the dead-letter queue.

For more information, see Overview of Service Bus dead-letter queues.

**Diagnostics**. Whenever the application moves a message to the dead-letter queue, write an event to the application logs.

# Service Fabric

### A request to a service fails.

**Detection**. The service returns an error.

**Recovery:**

- Locate a proxy again (`ServiceProxy` or `ActorProxy`) and call the service/actor method again.
- **Stateful service**. Wrap operations on reliable collections in a transaction. If there is an error, the transaction will be rolled back. The request, if pulled from a queue, will be processed again.
- **Stateless service**. If the service persists data to an external store, all operations need to be idempotent.

**Diagnostics**. Application log

### Service Fabric node is shut down.

**Detection**. A cancellation token is passed to the service's `RunAsync` method. Service Fabric cancels the task before shutting down the node.

**Recovery**. Use the cancellation token to detect shutdown. When Service Fabric requests cancellation, finish any work and exit `RunAsync` as quickly as possible.

**Diagnostics**. Application logs

# Storage

### Writing data to Azure Storage fails

**Detection**. The client receives errors when writing.

**Recovery:**

1. Retry the operation, to recover from transient failures. The retry policy in the client SDK handles this automatically.

2. Implement the Circuit Breaker pattern to avoid overwhelming storage.

3. If N retry attempts fail, perform a graceful fallback. For example:

    - Store the data in a local cache, and forward the writes to storage later, when the service becomes available.
    - If the write action was in a transactional scope, compensate the transaction.

**Diagnostics**. Use storage metrics.

### Reading data from Azure Storage fails.

**Detection**. The client receives errors when reading.

**Recovery:**

1. Retry the operation, to recover from transient failures. The retry policy in the client SDK handles this automatically.
2. For RA-GRS storage, if reading from the primary endpoint fails, try reading from the secondary endpoint. The client SDK can handle this automatically. See Azure Storage replication.

3. If *N* retry attempts fail, take a fallback action to degrade gracefully. For example, if a product image can't be retrieved from storage, show a generic placeholder image.

**Diagnostics**. Use storage metrics.

# Virtual machine

### Connection to a backend VM fails.

**Detection**. Network connection errors.

**Recovery:**

- Deploy at least two backend VMs in an availability set, behind a load balancer.
- If the connection error is transient, sometimes TCP will successfully retry sending the message.
- Implement a retry policy in the application.
- For persistent or non-transient errors, implement the Circuit Breaker pattern.
- If the calling VM exceeds its network egress limit, the outbound queue will fill up. If the outbound queue is consistently full, consider scaling out.

**Diagnostics**. Log events at service boundaries.

### VM instance becomes unavailable or unhealthy.

**Detection**. Configure a Load Balancer health probe that signals whether the VM instance is healthy. The probe should check whether critical functions are responding correctly.

**Recovery**. For each application tier, put multiple VM instances into the same availability set, and place a load balancer in front of the VMs. If the health probe fails, the Load Balancer stops sending new connections to the unhealthy instance.

**Diagnostics**. - Use Load Balancer log analytics.

- Configure your monitoring system to monitor all of the health monitoring endpoints.

### Operator accidentally shuts down a VM.

**Detection**. N/A

**Recovery**. Set a resource lock with `ReadOnly` level. See Lock resources with Azure Resource Manager.

**Diagnostics**. Use Azure Activity Logs.

# WebJobs

### Continuous job stops running when the SCM host is idle.

**Detection**. Pass a cancellation token to the WebJob function. For more information, see Graceful shutdown.

**Recovery**. Enable the `Always On` setting in the web app. For more information, see Run Background tasks with WebJobs.

# Application design

### Application can't handle a spike in incoming requests.

**Detection**. Depends on the application. Typical symptoms:

- The website starts returning HTTP 5xx error codes.
- Dependent services, such as database or storage, start to throttle requests. Look for HTTP errors such as HTTP 429 (Too Many Requests), depending on the service.
- HTTP queue length grows.

**Recovery:**

- Scale out to handle increased load.
- Mitigate failures to avoid having cascading failures disrupt the entire application. Mitigation strategies include:
  - Implement the Throttling pattern to avoid overwhelming backend systems.
  - Use queue-based load leveling to buffer requests and process them at an appropriate pace.
  - Prioritize certain clients. For example, if the application has free and paid tiers, throttle customers on the free tier, but not paid customers. See Priority queue pattern.

**Diagnostics**. Use App Service diagnostic logging. Use a service such as Azure Log Analytics, Application Insights, or New Relic to help understand the diagnostic logs.

### One of the operations in a workflow or distributed transaction fails.

**Detection**. After *N* retry attempts, it still fails.

**Recovery:**

- As a mitigation plan, implement the Scheduler Agent Supervisor pattern to manage the entire workflow.
- Don't retry on timeouts. There is a low success rate for this error.
- Queue work, in order to retry later.

**Diagnostics**. Log all operations (successful and failed), including compensating actions. Use correlation IDs, so that you can track all operations within the same transaction.

### A call to a remote service fails.

**Detection**. HTTP error code.

**Recovery:**

1. Retry on transient failures.
2. If the call fails after *N* attempts, take a fallback action. (Application specific.)
3. Implement the Circuit Breaker pattern to avoid cascading failures.

**Diagnostics**. Log all remote call failures.

## Next steps

For more information about the FMA process, see Resilience by design for cloud services (PDF download).