# Retry pattern

Azure

Enable an application to handle transient failures when it tries to connect to a service or network resource, by transparently retrying a failed operation. This can improve the stability of the application.

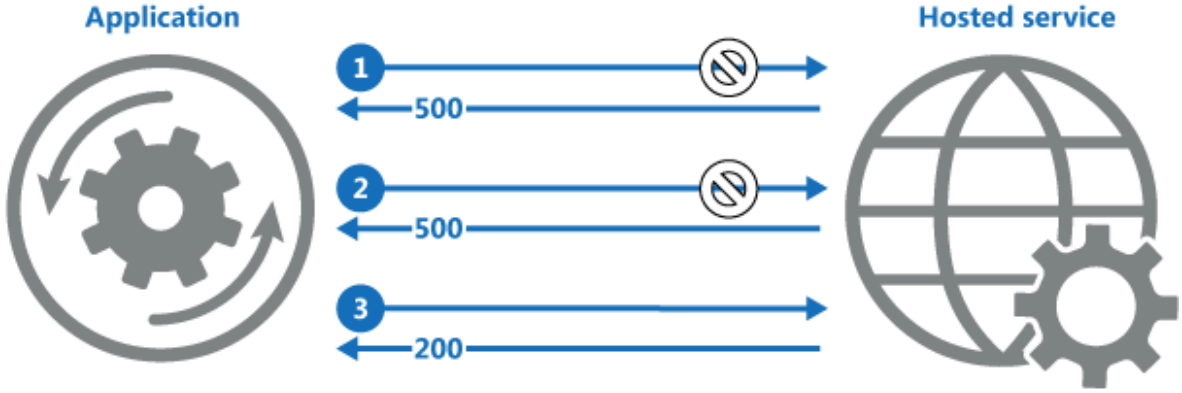
## Context and problem

An application that communicates with elements running in the cloud has to be sensitive to the transient faults that can occur in this environment. Faults include the momentary loss of network connectivity to components and services, the temporary unavailability of a service, or timeouts that occur when a service is busy.

These faults are typically self-correcting, and if the action that triggered a fault is repeated after a suitable delay it's likely to be successful. For example, a database service that's processing a large number of concurrent requests can implement a throttling strategy that temporarily rejects any further requests until its workload has eased. An application trying to access the database might fail to connect, but if it tries again after a delay it might succeed.

## Solution

In the cloud, transient faults should be expected and an application should be designed to handle them elegantly and transparently. Doing so minimizes the effects faults can have on the business tasks the application is performing. The most common design pattern to address is to introduce a retry mechanism.




1: Application invokes operation on hosted service. The request fails, and the service host responds with HTTP response code 500 (internal server error).
2: Application waits for a short interval and tries again. The request still fails with HTTP response code 500.
3: Application waits for a longer interval and tries again. The request succeeds with HTTP response code 200 (OK).

The diagram above illustrates invoking an operation in a hosted service using a retry mechanism. If the request is unsuccessful after a predefined number of attempts, the application should treat the fault as an exception and handle it accordingly.

> ⓘ Note
>
> Due to the commonplace nature of transient faults, built-in retry mechanisms are now available in many client libraries and cloud services, with some degree of configurability for the number of maximum retries, the delay between retries, and other parameters. The Microsoft Entity Framework provides facilities to retry failed database operations.

## Retry strategies

If an application detects a failure when it tries to send a request to a remote service, it can handle the failure using the following strategies:

- **Cancel**. If the fault indicates that the failure isn't transient or is unlikely to be successful if repeated, the application should cancel the operation and report an exception.
- **Retry immediately**. If the specific fault reported is unusual or rare, like a network packet becoming corrupted while it was being transmitted, the best course of action may be to immediately retry the request.
- **Retry after delay**. If the fault is caused by one of the more commonplace connectivity or busy failures, the network or service might need a short period while the

connectivity issues are corrected or the backlog of work is cleared, so programatically delaying the retry is a good strategy. In many cases, the period between retries should be chosen to spread requests from multiple instances of the application as evenly as possible to reduce the chance of a busy service continuing to be overloaded.

If the request still fails, the application can wait and make another attempt. If necessary, this process can be repeated with increasing delays between retry attempts, until some maximum number of requests have been attempted. The delay can be increased incrementally or exponentially, depending on the type of failure and the probability that it'll be corrected during this time.

The application should wrap all attempts to access a remote service in code that implements a retry policy matching one of the strategies listed above. Requests sent to different services can be subject to different policies.

An application should log the details of faults and failing operations. This information is useful to operators. That being said, in order to avoid flooding operators with alerts on operations where subsequently retried attempts were successful, it is best to log early failures as *informational entries* and only the failure of the last of the retry attempts as an actual error. Here is an example of how this logging model would look like.

If a service is frequently unavailable or busy, it's often because the service has exhausted its resources. You can reduce the frequency of these faults by scaling out the service. For example, if a database service is continually overloaded, it might be beneficial to partition the database and spread the load across multiple servers.

# Issues and considerations

You should consider the following points when deciding how to implement this pattern.

## Impact on performance

The retry policy should be tuned to match the business requirements of the application and the nature of the failure. For some noncritical operations, it's better to fail fast rather than retry several times and impact the throughput of the application. For example, in an interactive web application accessing a remote service, it's better to fail after a smaller number of retries with only a short delay between retry attempts, and display a suitable message to the user (for example, "please try again later"). For a batch application, it might be more appropriate to increase the number of retry attempts with an exponentially increasing delay between attempts.

An aggressive retry policy with minimal delay between attempts, and a large number of retries, could further degrade a busy service that's running close to or at capacity. This retry policy could also affect the responsiveness of the application if it's continually trying to perform a failing operation.

If a request still fails after a significant number of retries, it's better for the application to prevent further requests going to the same resource and simply report a failure immediately. When the period expires, the application can tentatively allow one or more requests through to see whether they're successful. For more details of this strategy, see the Circuit Breaker pattern.

### Idempotency

Consider whether the operation is idempotent. If so, it's inherently safe to retry. Otherwise, retries could cause the operation to be executed more than once, with unintended side effects. For example, a service might receive the request, process the request successfully, but fail to send a response. At that point, the retry logic might re-send the request, assuming that the first request wasn't received.

### Exception type

A request to a service can fail for a variety of reasons raising different exceptions depending on the nature of the failure. Some exceptions indicate a failure that can be resolved quickly, while others indicate that the failure is longer lasting. It's useful for the retry policy to adjust the time between retry attempts based on the type of the exception.

### Transaction consistency

Consider how retrying an operation that's part of a transaction will affect the overall transaction consistency. Fine tune the retry policy for transactional operations to maximize the chance of success and reduce the need to undo all the transaction steps.

## General guidance

- Ensure that all retry code is fully tested against a variety of failure conditions. Check that it doesn't severely impact the performance or reliability of the application, cause excessive load on services and resources, or generate race conditions or bottlenecks.
- Implement retry logic only where the full context of a failing operation is understood. For example, if a task that contains a retry policy invokes another task that also contains a retry policy, this extra layer of retries can add long delays to the processing. It might be better to configure the lower-level task to fail fast and report

the reason for the failure back to the task that invoked it. This higher-level task can then handle the failure based on its own policy.

- Log all connectivity failures that cause a retry so that underlying problems with the application, services, or resources can be identified.
- Investigate the faults that are most likely to occur for a service or a resource to discover if they're likely to be long lasting or terminal. If they are, it's better to handle the fault as an exception. The application can report or log the exception, and then try to continue either by invoking an alternative service (if one is available), or by offering degraded functionality. For more information on how to detect and handle long-lasting faults, see the Circuit Breaker pattern.

# When to use this pattern

Use this pattern when an application could experience transient faults as it interacts with a remote service or accesses a remote resource. These faults are expected to be short lived, and repeating a request that has previously failed could succeed on a subsequent attempt.

This pattern might not be useful:

- When a fault is likely to be long lasting, because this can affect the responsiveness of an application. The application might be wasting time and resources trying to repeat a request that's likely to fail.
- For handling failures that aren't due to transient faults, such as internal exceptions caused by errors in the business logic of an application.
- As an alternative to addressing scalability issues in a system. If an application experiences frequent busy faults, it's often a sign that the service or resource being accessed should be scaled up.

# Workload design

An architect should evaluate how the Retry pattern can be used in their workload's design to address the goals and principles covered in the Azure Well-Architected Framework pillars. For example:

Expand table

| Pillar | How this pattern supports pillar goals |
| --- | --- |
| Reliability design decisions help your workload become **resilient** to malfunction and to ensure that it **recovers** to a fully functioning state after a failure occurs. | Mitigating transient faults in a distributed system is a core technique for improving a workload's resilience.<br><br>- RE:07 Self-preservation<br>- RE:07 Transient faults |

As with any design decision, consider any tradeoffs against the goals of the other pillars that might be introduced with this pattern.

## Example

Refer to the Implement a retry policy with .NET guide for a detailed example using the Azure SDK with built-in retry mechanism support.

## Next steps

- Before writing custom retry logic, consider using a general framework such as Polly for .NET or Resilience4j for Java.
- When processing commands that change business data, be aware that retries can result in the action being performed twice, which could be problematic if that action is something like charging a customer's credit card. Using the Idempotence pattern described in this blog post can help deal with these situations.

## Related resources

- Reliable web app pattern shows you how to apply the retry pattern to web applications converging on the cloud.
- For most Azure services, the client SDKs include built-in retry logic.
- Circuit Breaker pattern. If a failure is expected to be more long lasting, it might be more appropriate to implement the Circuit Breaker pattern. Combining the Retry and Circuit Breaker patterns provides a comprehensive approach to handling faults.

## Feedback

Was this page helpful? Yes No