# Design for self healing

08/30/2018 • 4 minutes to read • Contributors

In this article



## Design your application to be self healing when failures occur

In a distributed system, failures happen. Hardware can fail. The network can have transient failures. Rarely, an entire service or region may experience a disruption, but even those must be planned for.

Therefore, design an application to be self healing when failures occur. This requires a three-pronged approach:

- Detect failures.
- Respond to failures gracefully.
- Log and monitor failures, to give operational insight.

How you respond to a particular type of failure may depend on your application's availability requirements. For example, if you require very high availability, you might automatically fail over to a secondary region during a regional outage. However, that will incur a higher cost than a single-region deployment.

Also, don't just consider big events like regional outages, which are generally rare. You should focus as much, if not more, on handling local, short-lived failures, such as network connectivity failures or failed database connections.

## Recommendations

**Retry failed operations**. Transient failures may occur due to momentary loss of network connectivity, a dropped database connection, or a timeout when a service is busy. Build retry logic into your application to handle transient failures. For many Azure services, the client SDK implements automatic retries. For more information, see Transient fault handling and the Retry pattern.

**Protect failing remote services (Circuit Breaker)**. It's good to retry after a transient failure, but if the failure persists, you can end up with too many callers hammering a failing service. This can lead to cascading failures, as requests back up. Use the Circuit Breaker pattern to fail fast (without making the remote call) when an operation is likely to fail.

**Isolate critical resources (Bulkhead)**. Failures in one subsystem can sometimes cascade. This can happen if a failure causes some resources, such as threads or sockets, not to get freed in a timely manner, leading to resource exhaustion. To avoid this, partition a system into isolated groups, so that a failure in one partition does not bring down the entire system.

**Perform load leveling**. Applications may experience sudden spikes in traffic that can overwhelm services on the backend. To avoid this, use the Queue-Based Load Leveling pattern to queue work items to run asynchronously. The queue acts as a buffer that smooths out peaks in the load.

**Fail over**. If an instance can't be reached, fail over to another instance. For things that are stateless, like a web server, put several instances behind a load balancer or traffic manager. For things that store state, like a database, use replicas and fail over. Depending on the data store and how it replicates, this may require the application to deal with eventual consistency.

**Compensate failed transactions**. In general, avoid distributed transactions, as they require coordination across services and resources. Instead, compose an operation from smaller individual transactions. If the operation fails midway through, use Compensating Transactions to undo any step that already completed.

**Checkpoint long-running transactions**. Checkpoints can provide resiliency if a long-running operation fails. When the operation restarts (for example, it is picked up by another VM), it can be resumed from the last checkpoint.

**Degrade gracefully**. Sometimes you can't work around a problem, but you can provide reduced functionality that is still useful. Consider an application that shows a catalog of books. If the application can't retrieve the thumbnail image for the cover, it might show a placeholder image. Entire subsystems might be noncritical for the application. For example, in an e-commerce site, showing product recommendations is probably less critical than processing orders.

**Throttle clients**. Sometimes a small number of users create excessive load, which can reduce your application's availability for other users. In this situation, throttle the client for a certain period of time. See the Throttling pattern.

**Block bad actors**. Just because you throttle a client, it doesn't mean client was acting maliciously. It just means the client exceeded their service quota. But if a client consistently exceeds their quota or otherwise behaves badly, you might block them. Define an out-of-band process for user to request getting unblocked.

**Use leader election**. When you need to coordinate a task, use Leader Election to select a coordinator. That way, the coordinator is not a single point of failure. If the coordinator fails, a new one is selected. Rather than implement a leader election algorithm from scratch, consider an off-the-shelf solution such as Zookeeper.

**Test with fault injection**. All too often, the success path is well tested but not the failure path. A system could run in production for a long time before a failure path is exercised. Use fault injection to test the resiliency of the system to failures, either by triggering actual failures or by simulating them.

**Embrace chaos engineering**. Chaos engineering extends the notion of fault injection, by randomly injecting failures or abnormal conditions into production instances.

For a structured approach to making your applications self healing, see Design reliable applications for Azure.