An introduction to designing reliable cloud services

September, 2012
An introduction to designing cloud services for reliability

This document is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EXPRESS, IMPLIED, OR STATUTORY, AS TO THE INFORMATION IN THIS DOCUMENT.

This document is provided “as-is.” Information and views expressed in this document, including URL and other Internet website references, may change without notice. You bear the risk of using it.

Copyright © 2012 Microsoft Corporation. All rights reserved.

The names of actual companies and products mentioned herein may be the trademarks of their respective owners.
Authors and contributors

MIKE ADAMS – Server and Tools Business
SHANNON BEARLY – Global Foundation Services
DAVID BILLS – Microsoft Trustworthy Computing
SEAN FOY – Microsoft Trustworthy Computing
MARGARET LI – Microsoft Trustworthy Computing
TIM RAINS – Microsoft Trustworthy Computing
MICHAEL RAY – Global Foundation Services
DAN ROGERS – Interactive Entertainment Business
FRANK SIMORJAY – Microsoft Trustworthy Computing
SIAN SUTHERS – Microsoft Trustworthy Computing
JASON WESCOTT – Microsoft Trustworthy Computing
# Table of contents

Overview ........................................................................................................................................... 1  
What is cloud service reliability? ........................................................................................................... 4  
Recovery-oriented computing .................................................................................................................. 6  
  ROC research areas ................................................................................................................................. 6  
Planning for failure ................................................................................................................................. 9  
  Core design principles for reliable services ............................................................................................... 11  
Designing for and responding to failure ............................................................................................... 13  
  Creating fault models ................................................................................................................................. 14  
  Designing and implementing coping strategies ....................................................................................... 15  
  Using fault injection .................................................................................................................................. 16  
  Monitoring the live site ............................................................................................................................... 17  
Summary .................................................................................................................................................. 18  
Additional resources ............................................................................................................................... 19  

Overview

This paper describes underlying reliability concepts and a reliability design and implementation process for organizations that create, deploy and/or consume cloud services. The paper will help equip the reader with a basic understanding of the fundamental concepts of reliability and can help decision makers understand the factors and processes that make cloud services more reliable. It can also provide architects, developers, and operations personnel with insights into how to work together to make the services they design, implement, and support more reliable.

The following figure highlights the spectrum of responsibilities between customers who purchase cloud services and the providers who sell them. For infrastructure as a service (IaaS) offerings, such as a virtual machines, responsibility is split between the provider and the customer. While the customer is responsible for ensuring that the solutions they build on the offering run in a reliable manner, the provider is still ultimately responsible
for the reliability of the infrastructure, (core compute, network and storage), components. When customers purchase software as a service (SaaS) offerings, such as Microsoft© Office 365, cloud providers hold primary responsibility for ensuring the reliability of the service. Platform as a service (PaaS) offerings, such as Windows Azure™, occupy the middle of this responsibility spectrum, with providers being responsible for everything the IaaS provider has, plus the OS (operating system) layer.

Figure 1. Cloud provider and cloud customer responsibilities

With the rise of cloud computing and online services, customers expect services to be available whenever they need them—just like electricity or dial tone. This capability requires organizations that build and support cloud services to focus on planning for probable failures and having mechanisms to rapidly recover from such failures. Cloud services are complex and have a number of dependencies, so it is important that all members of a service provider’s organization understand their role in making the service they provide as reliable as possible.

This paper includes the following sections:

- What is cloud service reliability?
- Recovery-oriented computing
- Planning for failure
- The process of designing for and responding to failure

Outside the scope of this paper, but also important to understand, is there are cost tradeoffs associated with some reliability strategies and these need to be factored into the decision about how to implement a service with the right level of reliability, and at the right cost. This could also entail what features to include in the service and prioritizing the degree of reliability associated with each feature.
What is cloud service reliability?

The Institute of Electrical and Electronics Engineers (IEEE) Reliability Society states that reliability [engineering] is “a design engineering discipline which applies scientific knowledge to assure that a system will perform its intended function for the required duration within a given environment, including the ability to test and support the system through its total lifecycle.”¹ For software, it defines reliability as “the probability of failure-free software operation for a specified period of time in a specified environment.”²

When applying these definitions to cloud services, organizations that create reliable software work to achieve the following goals:

- **Maximize service availability to customers.** Ensure that customers can access the service and perform the tasks that they need to perform to complete their work.

- **Minimize the impact of any failure on customers.** Assume that failures will occur but:
  - Minimize the impact a failure has on any given customer. For example, the service should be designed to degrade gracefully meaning that non-critical components of the service may fail but critical aspects still work.
  - Minimize the number of customers impacted by a failure. For example, by designing the service in such a way that faults can be isolated to one part of the service.
  - Reduce the number of minutes that a customer (or customers) cannot use the service in its entirety. For example, switching customer requests from one data center to another if a catastrophic failure occurs.

---

¹ IEEE Reliability Society, at [http://rs.ieee.org](http://rs.ieee.org)
² Ibid.
- **Maximize service performance and capacity.** Reduce the impact to customers when there is decreased performance, even if there is no detectable failure. For example, if an unexpected spike in traffic occurs for a service, instead of failing or providing a poor experience to all users, the service could be architected to give priority to paying subscribers versus free or trial users.

- **Maximize business continuity.** Focus on how an organization responds to failures when they happen. As much as possible, software and the associated services should be architected to handle large-scale disaster scenarios so the service will be recovered quickly and protect the integrity of the data where applicable. For services that cannot automate recovery, a disaster recovery plan for service restoration should be developed. In both cases, organizations and teams should conduct disaster recovery drills, including live failovers, to ensure they are prepared to respond quickly and efficiently when an actual failure occurs.

Recovery-oriented computing (ROC) provides an approach that can help organizations address each of these goals.
Recovery-oriented computing

Traditionally, systems have been architected with a focus on avoiding failures. However, the scale and complexity of cloud services brings inherent reliability issues. The ROC approach can help organizations frame software failure in a way that makes it easier to design cloud services to respond to these issues. There are three basic assumptions associated with recovery-oriented computing:

- Hardware will fail
- People make mistakes
- Software contains imperfections

Organizations that create cloud services must design them to mitigate these predictable failures as much as possible to provide a reliable service for their customers.

ROC research areas

ROC defines six research areas\(^3\) that can be adapted to cloud services design and implementation recommendations to mitigate potential issues that are rooted in the three basic assumptions. These six recommendations are explained in the following list:

- **Recovery process drills.** Organizations should conduct recovery process drills routinely to test repair mechanisms, both during development and while in production mode. Testing helps ensure that the repair mechanisms work as expected and do not compound failures in a production environment.

- **Diagnostic aids.** Organizations should use diagnostic aids for root cause analysis of failures. These aids must be suitable for use in

\(^3\) Recovery-Oriented Computing Overview, at http://roc.cs.berkeley.edu/roc_overview.html
non-production and production environments, and should rapidly detect the presence of failures and identify their root causes using automated techniques.

- **Fault zones.** Organizations should partition cloud services into fault zones so failures can be contained, enabling rapid recovery. Isolation and loose-coupling of dependencies are crucial elements contributing to fault containment and recovery capabilities. Fault isolation mechanisms should apply to a wide range of failure scenarios including software imperfections and human-induced failures.

- **Automated rollback.** Organizations should create systems that provide automated rollback for most aspects of operations, from system configuration to application management to hardware and software upgrades. This functionality does not prevent human error but can help mitigate the impact of mistakes and make the service more dependable.

- **Defense–in-depth.** Organizations should use a defense-in-depth approach to ensure that a failure remains contained if the first layer of protection does not isolate it. In other words, organizations should not rely on a single protective measure, but rather, factor multiple protective measures into their service design.

- **Redundancy.** Organizations should build redundancy into their systems to survive faults. Redundancy enables isolation so that organizations can ensure the service continues to run, perhaps in a degraded state, when a fault occurs and the system is in the process of being recovered. Organizations should design fail-fast components to enable redundant systems to detect failure quickly and isolate it during recovery.

Using the ROC approach can help an organization shift from strictly focusing on preventing failures to also focusing on reducing the amount of time it takes to recover from a failure. In other words, some degree of
failure is inevitable, (that is, it cannot be avoided or prevented), so it's important to have recovery strategies in place. Two terms can frame the shift in thinking that is required to create more reliable cloud services: mean time to failure (MTTF) and mean time to recover (MTTR).

MTTF is a measure of how frequently software and hardware fails, and the goal is to make the time between failures as long as possible. This approach works well for packaged software, because software publishers are able specify the computing environment under which the software will perform best. Cloud services require a different approach because portions of the computing environment are out of the control of the provider and thus more unpredictable. It is important, therefore, that cloud services are designed in such a way that they can rapidly recover.

MTTR is the amount of time it takes to get a service up and running again after a failure. Shrinking MTTR requires design and development practices that promote quicker detection and subsequent recovery, and it also requires well-trained operations teams which are capable of bringing components of the service back online as quickly as possible; better yet would be for the system itself to automatically recover. In addition, organizations should design cloud services so that they do not stop working, even when some subset of components fail; that is, the service can gracefully degrade while still enabling users to accomplish their work using that service. Embracing the ROC approach provides focus on design points that organizations can use to ensure that they design their services in a way that reduces MTTR as much as possible while continuing to increase MTTF as much as possible.
Planning for failure

To help reduce MTTR, organizations have a responsibility to plan for how their services will perform when known failure conditions occur. For example, what should the service do when another cloud service that it depends on is not available? What should the service do when it cannot connect to its primary database? What hardware redundancies are required and where should they be located? Can the service detect and respond gracefully to incorrect configuration settings, allowing rollback of the system? At what point is rollback of a given change no longer possible, necessitating a “patch and roll forward” mitigation strategy instead?

There are three primary causes of failure that must be considered by organizations creating cloud services. These causes are defined in the following figure:

Figure 2. Causes of failure

<table>
<thead>
<tr>
<th>Cause</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software imperfections</td>
<td>Code imperfections and software-related issues in the deployed online service. Pre-release testing can control this to a degree.</td>
</tr>
<tr>
<td>Human error</td>
<td>Administrator and configuration mistakes that are often out of an organization’s control.</td>
</tr>
<tr>
<td>Device and infrastructure failure</td>
<td>Ranging from expected, end-of-life failures to catastrophic failures caused by natural disaster or accidents that are out of an organization’s control.</td>
</tr>
</tbody>
</table>

Latent software bugs in a deployed service can cause performance to degrade over time or can cause the service to fail completely. During the service design stage, teams should design the instrumentation for the service to capture telemetry which can be used to diagnose possible failure conditions, particularly those that affect the customers’ experience.
Although systems and services can be designed to avoid direct change by humans, sometimes human intervention is unavoidable. One example of a common human error occurs when a person applies the correct configuration setting to the wrong component, or applies the wrong configuration setting to the correct component. Because well-intentioned and well-trained people will make mistakes, an organization must design cloud services to detect and compensate for such mistakes. For example, an organization may have server farms in its design, and it could include error detection and server isolation (sometimes referred to as logical circuit breakers) so that an initial misconfiguration does not replicate to other servers in the server farm. As another example, the organization could design the cloud service to roll out changes initially to a subset of the production environment and then perform automated monitoring to confirm that key performance indicators (KPIs) for the service are within specified levels before propagating changes out to the rest of the production environment. Highly mature services will automatically roll back these changes if KPIs vary from acceptable range, or will automatically continue to roll out changes if KPIs stay within an acceptable range.

Device and infrastructure failures range from regular, expected incidents such as routine, end-of-life failures, to those with huge impact but low frequency, for example as a result of a catastrophic event. For a cloud service to withstand these types of events, the service needs to be able to failover or use other techniques that save customer data and content across multiple physical locations, or have processes to recreate the data. For example, if power is lost at data center A, the service could be designed to quickly redirect traffic to data center B. Not only is it important that the traffic get redirected to data center B, but the service provider needs to also ensure the integrity of the data through adequate data protection techniques such as continuously replicating the data in A to B before the failure occurs.
Core design principles for reliable services

Organizations must address the following three essential reliability design principles when they create specifications for a cloud service. These principles help to mitigate the effect of failures when they occur:

- **Design for resilience.** The service must withstand component-level failures without requiring human intervention. A service should be able to detect failures and automatically take corrective measures so that users do not experience service interruptions. And, when failure does occur, the service should degrade gracefully, providing partial functionality rather than being completely offline. For example, a service should use fail-fast components and indicate appropriate exceptions so that the system can automatically detect and resolve the issue. There are also automated techniques that architects can include to predict service failure and notify the organization about service degradation or failure.

- **Design for data integrity.** The service must capture, manipulate, store, or discard data in a manner that is consistent with its intended operation. A service should preserve the integrity of the information that customers have entrusted to it. For example, organizations should replicate customer data stores so data will not be lost because of hardware failures, and adequately secure data stores to prevent unauthorized access.

- **Design for recoverability.** When the unforeseen happens, the service must be capable of being recovered. As much as possible, a service or its components should recover quickly and automatically. Teams should be able to restore a service quickly and completely if a service interruption occurs. For example, the organization should design the service for component redundancy and data failover so when failure is detected, whether it’s one component, a group of servers or an entire physical location or data center, the service
automatically uses another component, server(s), or physical location to keep the service running.

Organizations should adapt these essential principles as minimum requirements when they design their cloud services to handle potential failures.
Designing for and responding to failure

To build a reliable cloud service, organizations should create a design that specifies how a service will respond gracefully when it encounters a failure condition. The process that is illustrated in the following figure is intended for organizations that create SaaS solutions to help them identify possible failures and a process for mitigating those failures in the services they provide to their customers. However, organizations that purchase cloud services can also use this process to develop an understanding of how the services they subscribe to function and help formulate questions they should ask before they enter into a service agreement with a cloud provider.

Figure 3. An overview of the design process

- Create initial service design
- Create fault models
- Design coping strategies
- Use fault injection
- Monitor the live site
- Capture unexpected faults

Designing a service for reliability and implementing code that is based on that design is an iterative process. Design iterations are fluid and take into account both information garnered from pre-release testing and data.
about how the service is performing after it has been deployed to production.

Creating fault models

Creating a fault model for an online service is a key step in the design process. Identifying the important interaction points and dependencies of the service enables the engineering team to identify changes that are required to the design to ensure the service can be monitored and issues can be detected. This enables them to develop coping mechanisms so that the service is able to withstand or mitigate the fault. Fault models also help the engineering teams identify suitable test cases to validate that the service is able to cope with the fault both in test and in production (a.k.a. fault injection).

To create fault models, organizations should create a component inventory. This inventory includes all components that the service uses, whether they are user interface (UI) components hosted on a web server, a database hosted in a remote data center, or an external service that the service being modeled is dependent on. The team can then capture possible faults in a spreadsheet or other document and incorporate relevant information into design specifications. Some example questions that a team creating an online cloud service should address include:

- What external services will the service be dependent upon?
- What data sources will the service be dependent upon?
- What configuration settings will the service require to operate properly?
- What hardware dependencies does the service have?
- What are the relevant customer scenarios that should be modeled?

To fully analyze how the service will use its components, the team can create a matrix that captures which components are accessed for each
customer scenario. For example, an online video service might contain scenarios for logging in, for browsing an inventory of available videos, selecting a video and viewing it, and then rating the video after it’s been viewed. Although these scenarios share common information and components, each is a separate customer usage scenario, and each accesses some components that are independent from the other. The matrix should identify each of these usage scenarios and contain a list of all required components for each scenario.

Using a matrix also allows the service design team to create a map of possible failure points at each component interaction point, and define a fault-handling mechanism for each.

Designing and implementing coping strategies

Fault-handling mechanisms are also called coping strategies. In the design stage, architects define what the coping strategies will be so that the software will do something reasonable when a failure occurs. They should also define the types of instrumentation engineers should include in the service specification to enable monitors that can detect when a particular type of failure has occurred.

Designing coping strategies to do something reasonable depends on the functionality the service provides and the type of failure the coping strategy addresses. The key is to ensure that when a component fails, it fails quickly and, if required, the service switches to a redundant component. In other words, the service degrades gracefully but does not fail completely.

For example, the architects of a car purchasing service design their application to include ratings for specific makes and models of each car model type. They design the purchasing service with a dependency on another service that provides comparative ratings of the models. If the rating service fails or is unavailable, the coping strategy might mean the
An introduction to designing reliable cloud services

purchasing service displays a list of models without the associated ratings rather than not displaying a list at all. In other words, when a particular failure happens, the service should produce a reasonable result in spite of the failure. The result may not be optimal, but it should be reasonable, from the customer’s perspective. In our example it is reasonable to still produce a list of models with standard features, optional features, and pricing without any rating data rather than return an error message or a blank page, because the information that can be shown might be useful to the customer. Again, think in terms of “reasonable, but not necessarily optimal” when deciding what the response to a failure condition should be.

When designing and implementing instrumentation, it’s important to monitor at the component level as well as from the user’s perspective. This can allow the service team to identify a trend in component-level performance before it becomes a user-impacting incident. The data this kind of monitoring can produce enables organizations to gain insight into how to improve the service’s reliability for later releases.

Using fault injection

Fault injection is software designed to break software. For teams designing and deploying cloud services, it’s software designed and written by the team to cripple the service in a deliberate and programmatic way. It is often used with stress testing and is widely considered to be an important part of developing robust software.

When using fault injection on a service that is already deployed, organizations target locations where coping strategies have been put in place so they can validate those strategies. In addition, the cloud provider can discover unexpected results generated by the service and appropriately harden the production environment.
Fault injection and recovery drills can provide valuable information. These drills can reveal whether the service functions as expected or reveal unexpected faults that occur under load. A service provider can use this information to design new coping strategies to implement in future updates to the service.

**Monitoring the live site**

Accurate monitoring information can be used by teams to improve the service in several ways.

It can provide teams with information to troubleshoot known problems or potential problems in the service. It can also provide organizations with insights into how their services perform when handling live workloads. Lastly, it can also be fed directly into the service alerting mechanisms to reduce the time to detect problems and therefore reduce MTTR.

Simulated workloads in a test environment rarely capture the range of possible failures and faults that live site workloads generate. Organizations can identify trends before they become failures by carefully analyzing live site telemetry data and establishing thresholds, both upper and lower ranges, that represent normal operating conditions. If the telemetry being collected in near real time approaches either the upper or lower threshold, an alarm can be triggered, prompting the operations team to immediately triage the service and potentially prevent a failure. They can also analyze failure and fault data that instrumentation and monitoring tools capture in the production environment to better understand how the service operates and to determine what monitoring improvements and new coping strategies they require.
Summary

To design and implement a reliable cloud service requires organizations to assess how they regard failure. Historically, reliability has been equated with preventing failure—that is, delivering a tangible object free of faults or imperfections. Cloud services are complex and have dependencies, so they become more reliable when they are designed to quickly recover from unavoidable failures, particularly those that are out of an organization’s control. The processes that architects and engineers use to design a cloud service can also affect how reliable a service is. It is critical that service design incorporates monitoring data from the live site, especially when identifying the faults and failures that are addressed with coping strategies tailored to a particular service. Organizations should also consider conducting fault injection and recovery drills in their production environments. Doing so generates data they can use to improve service reliability and will help prepare organizations to handle failures when they actually occur.
Additional resources

- “To Err is Human,” Brown and Patterson (PDF) http://roc.cs.berkeley.edu/papers/easy01.pdf