Windows Azure Best Practices

1. Validate your approach in the cloud early on. Once you have a tentative solution design, test the viability of your design in the cloud early in your project to validate its viability. Don't wait until the end of your project to find out if your approach will work. Your solution in the cloud may reveal very different dynamics and performance from the enterprise, and you may encounter issues with supported features and deployment. Waiting until the last minute to test viability in the cloud is dangerous.

2. Run a minimum of 2 server instances for high availability. When hosting in Windows Azure, consider 2 as the minimum server farm size. If you only run 1 server, you will experience periods of unavailability as patches or updates are applied. When you have 2 more servers in a farm, patches and updates are sequenced to prevent downtime. The Windows Azure SLA is only guaranteed when there are at least 2 servers in a farm.

3. SOA best practices apply to the cloud. The principles of Service-Oriented Architecture apply equally well to the cloud as they do in the enterprise:

- Connect programs with loose coupling
- Define explicit boundaries
- Use interoperable standards
- Define coarse-grained contracts and avoid chatty interfaces
- Favor stateless services

4. Be as stateless as possible. Farms for web applications, web services, and workers can be adjusted at any time to larger or smaller sizes. The best way to code wherever possible is to avoid session state. If you must have session state, implement it in a way that doesn't rely on server affinity.

5. Co-Locate code and data as much as possible. Whether your code is in the cloud or in the enterprise, it's best if code and the data it accesses are near each other. Avoid unnecessary traffic between cloud and enterprise.

6. Take advantage of data center affinity. Since many cloud solutions involve multiple projects (hosting, storage, database, communication, security) use the Affinity Group feature of the Azure management portal to ensure a related set of projects reside in the same data center.

7. Retry calls to Windows Azure services, SQL Azure databases, and your own web services before failing. Any HTTP call can fail, and you should expect calls to the cloud will fail or time out at times. Your calling code should attempt a limited number of retries before failing. This applies to your own web services, to Windows Azure services (storage, communication, security), and to SQL Azure databases.

8. Use separation of concerns to isolate cloud/enterprise differences. Use common interfaces and dual service implementations to provide equivalent cloud and on-premise functionality to your apps. If you're writing applications that need to work in the cloud as well as on-premise, you'll have to implement some things slightly differently—for example, accessing a local file server vs. accessing blob storage in the cloud. A great way to do this is to define a common service interface (with functions like GetFile, PutFile) and have two implementations, one for on-premise and one for in the cloud.

9. Get as Current As Possible Before Migrating to Azure. The more current you are on Microsoft technologies in your enterprise app before migration, the easier the migration will be. A case in point is SQL Server to SQL Azure. If you're not on SQL Server 2008 yet, you should get compatible with that before migrating to SQL Azure. Otherwise you'll have twice the number of issues in getting the database migrated and it will be less clear what's an Azure issue vs. a SQL Server version issue.

10. Migrate applications one tier at a time. Since most solutions have multiple tiers, a migration to the cloud can be approached one tier at a time—keeping the solution running all the while. This is analogous to keeping a patient alive through a series of surgeries. By focusing on one software layer at a time, you carve a complex job into a series of smaller more focused tasks. It's best to migrate database first, and then web services, then front end.

11. Understand the security requirements of the service being designed or migrated. Security requirements especially in the context of authentication, authorization and auditing have to be understood. The platform services which provide features (Windows Identity Foundation, Windows Azure AppFabric Access Control Service, Windows Azure Monitoring and Diagnostic APIs) and the methods developers use to invoke them are substantially different from those provided in an on-premise enterprise deployment (Kerberos, Active Directory, Windows Event Logs).

12. Apply SDL Practices to Windows Azure Applications. The Microsoft Security Development Lifecycle (SDL) applies equally to applications built on the Windows Azure platform and any other platform. Most Windows Azure applications have been built, or will be built using agile methods. As a result, the SDL for agile process may be more applicable to applications hosted on Windows Azure than to the classic phase-based SDL.

13. Check Request Throttling / Input Sanitization. Developers must do application-level throttling of incoming requests for any kind of complex, time-intensive operation. The Microsoft Security Development Lifecycle (SDL) portal provides resources on fuzzing parsers. If a service is parsing a proprietary file or request format (perhaps encapsulated inside HTTP), then fuzz test it to ensure the code can correctly accommodate malformed input.

14. Use the "Gatekeeper" design pattern to separate role duties and isolate privileged access.
A Gatekeeper is a design pattern in which access to storage is brokered so as to minimize the attack surface of privileged roles by limiting their interaction to communication over private internal channels and only to other web/worker roles. These roles are deployed on separate VMs. In the event of a successful attack on a web role, privileged key material is not compromised.