**ICON Test( Client Application upgrade in Azure Cloud)**

**Table of contents:**

**ICON Client Application Upgrade Architecture** - - - - - - - - - -- - - - - - - -- - - Page 2-3

**Current Infrastructure** ------------------------------------------------------------------ Page 2

**Proposed Infrastructure** ---------------------------------------------------------------- Page 3

**Architecture Diagram** ------------------------------------------------------------------- Page 3

**Terraform components** ----------------------------------------------------------------- Page 4

**CICD strategy** ------------------------------------------------------------------------------ Page 5

**ICON Application Upgrade Architecture:**

The issue Client is facing with the current Application hosted Architecture is that the application response is becoming increasingly slow during peak business hours, the client is looking for a solution to upgrade it’s application’s infrastructure so that the application’s response remains same during the peak business hours as the non-peak business hours response is.

**Current infrastructure:**

1. App service plan for production website is S2

(Standard Plan)

1. Database instance in an Icon shared Database infrastructure.
2. 1 VM with exposed Endpoint which application access.
3. Classic Blob storage used to upload files from the production website.

**Proposed Infrastructure:**

1. Website to be hosted in App service Plan P2 for Production and Staging Website.

(Premium Plan). To include Azure app service autoscaling feature so that in peak business hours the application will be able to scale horizontally and cool down once the peak business hours are over.

1. Use of separate Azure SQL database server for Production and staging applications to isolate the client’s databases.
2. To make the database operations more optimized we are going to use a in memory Redis Cache, We are using Azure Cache for Redis service for the in memory Cache.

Azure Cache for Redis is fully managed service that is provided by Azure so we don’t have to worry about the autoscaling, Reliability and availability of the Redis Cache as all of this is the responsibility of Azure Cloud.

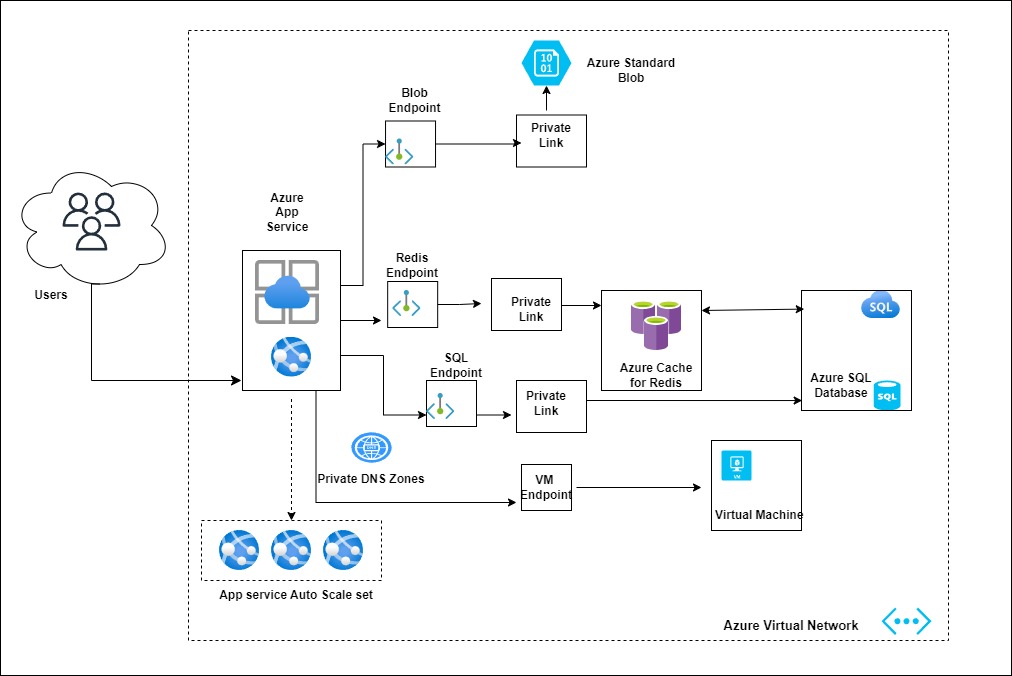
We are using Redis cache in this solution so that in the peak hours when the application’s query operations for the Database increases drastically and hence the database response becomes slow that affects the overall performance of the application is taken care off.

When we use Redis Cache the most queried contents of the database will be cached in Redis Cache hence it will be faster, and less load will be applied to the SQL database itself.

**Note: To integrate Redis Cache in this Architecture we will need to amend Application code so that it will cache contents for SQL Database in Redis.**

1. In order to make the website response better and faster during peak hours we are going to implement autoscaling feature in the azure app service because of which when the load is increased on the website during peak hours the application will be able to scale out horizontally to the maximum instances of the application without the need for manual intervention, and once the load is decreased during off-peak hours the application will be able to scale in horizontally to the minimum instances of the application running. Hence the application performance will remain constant and will not degrade in Peak business hours.
2. We are going to replace the Classic Blob storage with the Azure Standard Blob storage.
3. We Will create a Virtual machine in the same virtual network which website will access through an exposed endpoint.
4. We will be using private Endpoint for azure SQL, Azure Blob Storage, Azure Redis for Cache, Azure Windows Virtual Machine. So that the website deployed on Azure App service Communicates with all the mentioned services using Azure Back Bone Network and not through the Internet.

**Upgraded Application Architecture Diagram**



Note: we will follow the same application architecture for production and staging environment

**Terraform:**

To deploy the Infrastructure of Client’s Application on Azure Cloud We are using Terraform as IaC tool. We can Use the same template to deploy Production and Staging Applications.

In Terraform We are Creating the following resources

1. 1 Resource Group for the whole application components.
2. 1 Virtual Network.
3. 5 subnets. ( App-subnet, Redis-subnet, sql-subnet, vm-subnet, and asa-subnet)

* App-subnet for app service.
* Redis-subnet for Redis Cache private Endpoint.
* Sql-subnet for sql Private Endpoint.
* Vm subnet for virtual machine private Endpoint
* Asa-subnet for Blob storage Endpoint.

1. The “**appservice.tf**” file is used for the terraform deployment code for azure App service Plan, App service and its VNet integration.
2. The **“appserviceautoscale.tf”** is used for the Terraform deployment of App service Auto-Scaling so that once the CPU usage for the app service reaches >80% it scales the App service instances by 1 and cool it back down once the <50.
3. The **“azurerediscache.tf”** is used to create an Azure Cache for Redis instance and its private endpoint through which we can have a connection for the Azure App service to the Redis Cache.
4. The **“azuresqlserver.tf”** is used to create azure SQL and a Database named “Production\_db” and the SQL private endpoint.
5. The **” blob.tf”** is used to create standard Blob storage and its private endpoint.
6. The **“virtualmachine.tf**” is used to create a Windows Server Virtual machine and a Private Endpoint.
7. The **“network.tf”** file is used to create a Virtual Network, Subnets for Azure SQL private Endpoint, Azure Cache for Redis private Endpoint, Azure Blob Storage Private Endpoint and Azure Virtual Machine and an Azure Virtual Machine Private Endpoint.
8. The **“provider.tf”** contains configurations for the Terraform Provider, in our case it is Azure.
9. **“variables.tf”** and **“variables.tfvars”** are used for environment variables.

Note: The Terraform Template is not tested in the cloud environment so may require some tweaks.

**Continues Integration and Continues Deployment Strategy:**

1. For the CICD of the application we will use Azure DevOps Build and release pipelines.
2. The Azure DevOps build Pipelines will run on Azure Self hosted Agents
3. **Self-Hosted Agents:** We will create a virtual machine in Azure. We will deploy azure self-hosted agent on that virtual machine so that it will work with the azure pipelines. We will add all necessary packages required in the application build stage on the virtual machine.
4. **Build Pipeline:** Add the GitHub repository of client as source in azure build pipeline.
5. Add the self-hosted agent in the build pipeline
6. We will define the necessary tasks needed to complete the build processes.
7. **Triggers:**

**Production:** In production Build pipeline we will set the trigger to enables continues integration for branch filter we will add the Master branch as it is required to trigger production build once a change is committed in the master Branch.

**Staging:** In Staging Build pipeline we will set the trigger to enables continues integration for branch filter we will add the Staging Branch as it is required to trigger staging build once the change is committed in Staging branch.

1. We will configure the build number format for the build pipeline.
2. **Release Pipelines:** In the release pipeline the artifact will be used be used for staging and production build pipeline.
3. Create a service connection in the Azure DevOps for the Azure App service
4. Use the web app service deployment task in the release pipelines for the continues deployment.
5. In the Release Pipeline use Pre-Deployment approvals, add necessary users for the project manager and CTO in the pre deployment approvals for both staging and production release pipelines.
6. Ideally set All web app settings as Environment variables in release Pipelines.