First of all, I designed a solution as simple as possible. When we start to complicate things, normally they don’t progress and we face ourselves complicating the solution even more. So, keep things simple was my first taught when drawing this solution.

About the network, we have a vNet with a CIDR 10.16.0.0/16 and three subnets. One public with CIDR 10.16.10.0/24 that has two things inside it. An Application Gateway that will expose microservices to internet and a Virtual Machine that will be the bridge for DEVs and SRE engineers to access production environment. We will have a private subnet with CIDR 10.16.11.0/24 for the AKS cluster and a second private subnet with CIDR 10.16.12.0/24 for CosmosDB and Postgres. AKS, Postgres and CosmosDB won't have public links at least in production environment. Another thing that deserves a mention is that all the incoming request will pass through a WAF shield that ensure security by default

Now I will explain my choices about the Azure services, so let’s go through the topics.

**Variables Loads & JS Front End**

Nowadays this is a big problem for companies. Specially E-Commerces. They don’t have heavy loads all year. The count of requests increases near holidays.

To solve this problem in my design, I’m using a great solution of Microsoft called Azure CDN (Content Delivery Network). This service provides a world-wide solution to deliver our static website files to our clients. This service can be multi-region, so a client that is based on EU will have a great latency experience when accessing our website. The same with a client based in South America. So this is the service that I chose to deliver our website. And it has integration with Azure Storage Account that will store our website static files.

Now, to solve this problem for our microservices APIs, I choose AKS. We have great benefits from it. AKS is an acronym for Azure Kubernetes Services and it provides a Kubernetes cluster for our APIs. Kubernetes from itself already solves the **Variable Loads** problem because we can configure our app deployment with an HPC (Horizontal Pod Autoscale). This configuration will tell Kubernetes to scale up our APIs horizontally when the Pod CPU reaches more than 70% of utilization for example. So we can scale our APIs using Kubernetes very easily.

**Collection of containerized microservices**

For this topic, I chose Kubernetes because it is a container orchestration platform. Kubernetes reduce operation overhead because:

* It is always looking for containers health;
* Drop containers when they crash and run a new one in his place;
* Scale up containers when is needed and also scale down when the there's no need anymore.
* You can configure an *init container*, for example, load a file from a web api, load data to a database api need or run database schema migration
* You can have a logical separation of contexts using namespaces. For example, a namespace called *market-invetory* that have 4 microservices on it and other namespace called market-checkout that has 2 microservices. This feature helps devs when they need to see how is their microservices health, get some logs and a lot of other things.

There's uncountable possibilities for any kind of need on Kubernetes.

**Postgres and MongoDB**

For these two guys, I chose Azure Postgres for the Postgres database and CosmosDB for the MongoDB. Why CosmosDB? Because it is a very consolidate solution when you need a NoSQL database using MongoDB APIs and it has great features like multi-region data replication and high availability.

**Optimizing for Cost**

The design is already saving costs when we choosed storage account to store static website files. It's pretty cheap.

Another possible action to save costs is enable AKS cluster auto-scale to increase cluster nodes when needed.