International Journal on Science and Technology (IJSAT)
E-ISSN: 2229-7677 ● Website: www.ijsat.org ● Email: editor@ijsat.org

Designing Highly Available and Scalable Web Applications Using Azure App Services and Azure Functions

Anil Kumar Manukonda

anil30494@gmail.com

Abstract

Web applications today must be able to support and serve users around the world without any interruptions. This paper describes a way to design architecture by using Azure App Services and Azure Functions to satisfy these demands. We outline the goals and approaches for developing web applications on the cloud that can deal with sudden application stops. It is clear from our study that using Azure's functionality, like load balancing, auto-scaling and multi-zone deployment, leads to almost continuous uptime and helps cope with any sudden increase in traffic. We use Azure App Service for the front end and Azure Functions based on Python for the back end, setting up everything using Terraform as code. They reveal that scaling horizontally results in much better throughput than scaling vertically. We discuss the high availability services such as Azure's 99.95% SLA guarantees, Traffic Manager and Front Door. The comparison table makes it clear that using App Service all the time is more expensive than using Azure Functions only when needed. Steps for identity management using Azure AD (Entra ID), secure storage of secrets in Key Vault and a safe DevOps process are recommended. The concepts are demonstrated with a live case study of the Contoso retail app. To sum up, the paper points out that using both Azure App Services and Functions in combination enables web applications to be highly scalable, always available, safe, cost-efficient and in line with today's best cloud-based practices for enterprises.

Keywords: Azure App Services, Azure Functions, High Availability, Scalability, Serverless Architecture, Load Balancing, Auto-Scaling, Availability Zones, Multi-Region Deployment, Azure Front Door, Azure Traffic Manager, Terraform, Infrastructure as Code, Event-Driven Execution, Consumption Plan, Premium Plan, Function Triggers, Horizontal Scaling, Vertical Scaling, Performance Benchmarking, Application Insights, Cost Optimization, Azure Key Vault, Managed Identity, Microsoft Entra ID, Secure Deployment, DevSecOps, CI/CD Pipelines, Disaster Recovery, Active-Passive Failover, Active-Active Deployment, Azure Monitor, Application Gateway, Azure Storage Queue, Azure Cosmos DB, Azure SQL Database, Azure AD B2C, Web Application Firewall

Introduction

It is very important in modern development to consider the challenge of scalability and availability. With more people using the internet and fluctuating app usage, applications should automatically grow or shrink to maintain their performance and keep downtime low. If a web

application fails to grow, it can decrease its growth and negatively impact users and the company. Outages are also costly according to Gartner, with the average downtime cost coming out to $5,600 each minute or $350k every hour. That's why it's vital to focus on HA to please customers and safeguard business profits [7].

By using Microsoft Azure, businesses can handle such issues because the platform is always ready with available resources and redundancy. Azure attracts users with its global data centers and the services that help minimize complex tasks related to scaling and resuming operation after failures. So, Azure App Service (as a Platform as a Service for web apps) has the ability to scale services as necessary and position them in data centers around the global [1]. Developers can meet strict time targets because App Service and Function Apps in Azure are guaranteed to work 99.95% of the time in a specific region [6]. Employing cloud-native practices such as designing services that do not hold data, using microservices or functions and automating their infrastructure, allows organizations to become more scalable and reliable than they could be inside their own data centers.

This study explores how using Azure App Services and Azure Functions improves the scalability and availability of web systems. We explain that in cloud-native design, we break the application into separate services or functions, utilize managed databases and caches and depend on Azure to handle load balancing and recovery in case of errors. This section covers Azure App Service and Functions in detail, gives a sample plan for serverless web apps (including Terraform scripts and code to run applications), measures performance both with and without auto-scaling, explains strategies for high availability, optimizes costs, discusses security and includes a case study (Contoso's retail app) that demonstrates all these approaches. The aim is to provide architects and developers with a strong yet useful plan from Azure's cloud services that ensures web applications will scale horizontally and still be highly available even in difficult situations.

Technical Overview

This section discusses how Azure App Services and Azure Functions are designed and outlines how main Azure tools such as load balancing, auto-scaling and availability zones improve both scaling and availability.

Azure App Service Architecture:

You can use Azure App Service to host website applications, APIs and services without needing to manage the servers. Within an App Service plan, each App Service app receives VM resources, price level, its region and the number of instances assigned to it. This happens automatically without the need for additional actions from the user. Developers distribute their code to Azure in a popular language and Azure will ensure the rest is taken care of [1]. As a result, teams can pay more attention to their apps and less to matters of virtual machines or scaling.

Once an app is deployed in Azure, it will be assigned to worker instances. All of the incoming HTTP requests in App Service are handled evenly by the built-in load balancer. As a result, each application has enough capacity and nothing becomes overloaded. All Facades will be accessible using the same URL and will use the same app endpoints (optional) [6]. More instances can be set up if there is more demand; according to Part of the System, these can be removed when demand decreases. Scale out and scale in tasks can be accomplished either by hand or through Azure's Autoscale which acts when its rules are met. As a result, you may decide to set Add an instance if the CPU is above 70% for

5 minutes and remove one when it is below 30%. Thanks to Azure Monitor, it is simple to build rules in Azure App Service based on CPU, memory, the queue for user requests and other metrics.

Different instances can be supported in each App Service tier and the chosen tier also determines the capacity of each instance. You can scale beyond a single plan's limit by making copies of the app in different plans and having an external tool load balance (or distribute traffic) among them. This happens mainly when more than one plan is needed for vital applications, so the instances are not mixed and can be isolated from influence [2]. If this happens, you would use Azure Front Door or Traffic Manager to help with various tasks (explained in the High Availability section).

It is possible to scale vertically (scale-up) by selecting a larger type of instance to deal with a heavier workload. Changing to a higher tier on the pricing plan is simple, yet the app might have to be restarted for a moment. Instead of that, scaling horizontally (scale-out) includes more copies of the present machine. Usually, web workloads use horizontal scaling because the process is automated and one VM's hardware is not a problem. According to Azure, applying autoscaling and ensuring there is some extra capacity helps avoid drops in performance while scaling [2]. With horizontal scaling, if something happens to one instance, other instances will keep working, but with a purely vertical solution, one trouble instance can shut everything down.

Azure App Service can run an app in multiple Availability Zones to ensure greater reliability within a given region. An availability zone refers to a sectioned data center powered and connected on its own. In most cases, App Service places instances across different racks but not certain zones. When aiming for increased reliability, you can use zone redundancy (available with Premium v3 or more) to ensure your instances are always in separate zones in the same region. In that case, the app would continue running in the functional zones [3]. They need to be installed in a minimum of three environments (for each zone). When a zone-redundant deployment is used, Azure ensures that traffic goes to working machines in each zone. So, you achieve high availability within the region and better fault tolerance (even though Azure's SLA for App Service is not changed). As shown in Figure 1, an App Service is active in more than one location and is handled by Front Door, a global load balancer. Within each region where App Service runs, set it zone-redundant for even higher protection.

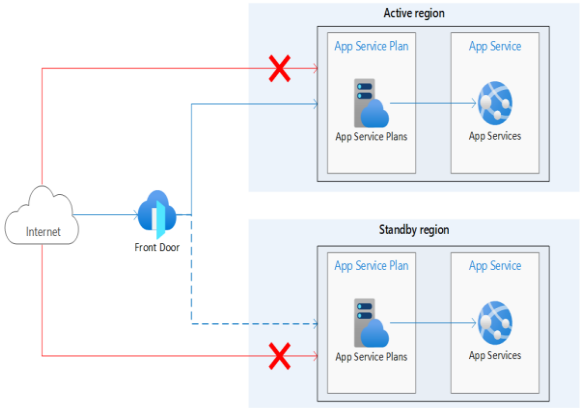

Figure 1: Multi-region Active-Passive deployment using Azure Front Door to route users to an Azure App Service in the primary region, with failover to a secondary region. The red X next to the endpoint shows that you are using Front Door for all traffic. In addition, multiple Availability Zones in the region can be utilized by an App Service Plan to ensure greater reliability [2].

Overall, Azure App Service ensures web apps can grow easily by giving them an environment for many customers, automatic load balancing and options for scalable size changes. Azure takes care of handling the amount of resources needed and the traffic directed through the traffic distributor. Laying out the environment like this allows you to set up highly available services, mainly owing to using cross-region services and other Azure tools.

Azure Functions Architecture:

Azure Functions is Azure's serverless service that offers FaaS (Functions-as-a-Service) for running small chunks of code. Azure Functions relies on and uses Azure App Service, however, it hides these details behind the scenes. It is up to you to deploy separate modules of code and Azure will start them, handle their scaling when needed and remove the environment when everything is idle.

An Azure Function App allows you to run one or several functions. Basically, a Function App based on the Consumption Plan resides on a pool of VMs handled by Azure. Whenever events appear, Azure provides a worker to execute the function. If the server receives many requests at the same time, the system will automatically generate new instances to handle them. The auto scaling makes it possible for the Function App to add up to 200 instances very fast and will remove those that have not been used in a period – eventually scaling to zero again [10]. Because of this, Azure Functions on a consumption plan grows to handle any load in seconds with no extra input from the user. Some triggers allow an instance to process multiple executions at once to handle more events.

It should be mentioned that Azure Functions is an event-driven scaling technology. Azure will handle incoming requests of HTTP-triggered functions by adding more instances, but for queue-triggered functions, the platform scales based on the queue's length and how fast functions are executed. Triggers differ in concurrency as well since HTTP scales every thousand pending requests to a different function, while Event Hubs may let a trigger handle batches of events at a time. The scale controller in Azure Functions watches the dealing events and determines the need for additional or fewer threaded instances. These benefits are possible due to Azure automatically running scale rules for the developer which is an advantage of serverless services.

For architecture, running your code with the Consumption plan allows Azure to reuse your containers. Sometimes, loading your code takes longer at the start for a new instance (mainly if the function has been unused). If low-latency functions must perform quickly, Azure offers both a Premium Plan and the more recent Flex Consumption Plan to reduce cold starts. The Premium Plan warms up a particular number of instances so they are ready and also provides chances for using high-resource VMs and allowing VNet usage. The Preview version of Flex Consumption provides you with options to specify the base number of available instances and the resource capacity per each one. You can run your Functions on a regular App Service Plan (dedicated VM) for longer cases, but they will function just like a standard App Service with costs that do not change automatically [10]. The paper concentrates on the Consumption model since it is the most advantageous for using Azure Functions in terms of scaling.

Azure Functions gives the option to run functions in several availability zones (for selected plans) and spread across regions to ensure disaster recovery. App Service is similar to how reliability works with Azure. Additionally, a single region in Azure Functions comes with an SLA of 99.95% [6]. Ensuring high availability involves having identical functions running in 2 different regions and having Traffic Manager or Front Door forward any trigger for the primary region to the secondary region (a

method called active-passive failover or event sourcing) if the primary crashes.

In truth, Azure Functions often acts as the foundation for your application: whenever needed, you can add HTTP-triggered capabilities and functions for various background activities. Because they function apart, applying these functions helps the web app to handle more workload more efficiently. In our example and case study, the web application uses Azure Functions to complete certain responsibilities.

Overall, Azure Functions has event-driven execution, automatic scaling from zero instances to many instances as needed and a pay-as-you-use model. It invites working with microservices or micro-functions, where each function is responsible for one task. This follows the principles of cloud-native designs and can be paired with Azure App Service when building a web application with features on the front-end and features running in the background.

Azure Load Balancing and Traffic Distribution:

Azure Load Balancer works at layer 4 and divides incoming requests among several virtual machines. Although Azure Load Balancer often works with virtual machine scale sets and container services, App Service and Functions already use load balancing within their platform functions. But, familiarizing yourself with Azure's load balancing is necessary when creating external traffic workflows and multi-region networks. To implement the "stamps" pattern using App Service, we would have Azure assign traffic to all the identical deployments.

Azure gives you several managed load-balancing services such as Azure Load Balancer, Azure Application Gateway, Azure Traffic Manager and Azure Front Door. In a region, Application Gateway balances and sends traffic among services (if we also want advanced routing or some security with Web Application Firewall, we put it in front of App Service). However, Traffic Manager and Front Door do this job across regions.

Notably, the front-end on Azure App Service distributes all incoming HTTP(S) requests among the instances in the same App Service Plan using load balancing [6]. Everything is set up automatically, so users do not need to adjust settings – it gives load balanced traffic and prevents sending to instances that do not work correctly. As a result, if an app is set up for N instances within a plan, its balance is already set for higher availability among those group instances.

To achieve multi-region high availability, Azure Traffic Manager and Front Door can be used (more details will appear in the High Availability Strategies section). Overall, Traffic Manager uses DNS for routing: it gives a DNS name to several targets (such as App Services in Regions A and B) and routes clients using different methods (such as priority, where a client is located or weighted). If the main endpoint isn't responding, Traffic Manager can set up the secondary as the next choice for new DNS requests. Although transferring to another DNS server can fail during the TTL time, so things can take a little while. Azure Front Door, different from the others, is an HTTP(s) load balancer that works near the edge of Microsoft's network. It provides faster failover, can terminate SSL, route traffic according to specific paths and act as a CDN. Front Door rapidly identifies the fastest endpoint and automatically takes down unreachable areas in minutes. Our architecture design suggests using Front Door to balance the load and enhance speed for a web app since it offers local POPs in various regions.

Auto-Scaling in Azure:

Without auto-scaling, it would be very difficult to build an elastic workload. App Service and Azure Functions differ in the way they provide automatic scaling. For App Service (using dedicated plans), you must create Autoscale goals using Azure Monitor. This rules are used to decide when to start or end the process of scaling. If, on average, the CPU exceeds 70% for 5 minutes, add an extra instance (up to a max of 5 instances); on the other hand, if it goes below 30% for 10 minutes, scale down by one instance (a minimum of 2 instances need to be maintained). The App Service Plan capacity will be increased or decreased by Azure's autoscale engine based on the above information [11]. Most campaigns are more reliable if you still have some spare space before reaching your final user load. Autoscale can also be set to act on your behalf such as adding 10 instances every day at noon when traffic is expected to rise. Azure APP Service's autoscale happens when it detects a surge, but you can still identify appropriate benchmarks for sudden traffic using load tests.

On the Consumption plan, Azure Functions can handle autoscale by itself after it detects certain events. Scaling resources up and down is one reason many people are interested in Functions. Still, the Premium/Flex plans allow you to set the minimum, maximum and concurrent values for your application instances. It is possible to put a cap on the maximum scale for Consumption plan function apps, though by default the limits are very high. In tests, Azure Functions are capable of handling dozens of instances in only seconds. When more than 10,000 requests per second happened, the function app was able to scale up to 80 processes in about 10 seconds. The extreme test of the Flex Consumption plan shows that a function app can handle more than 975 instances at the same time and supply near 40,000 requests per second. It shows how easily Azure's serverless service can handle increased traffic [9].

An application's design should include plans for which components can be made to scale out. If the load is sustained, App Service will use a fixed number of instances and let autoscale rules increase or decrease them gradually. Azure Functions is able to increase its capacity quickly when needed. Combining methods is helpful: the App Service handles smaller tasks consistently for users and the main processing is saved for Functions that can be used in very large numbers. Owing to this approach, interactive workloads do not interact with the surge loads in another area.

Azure users can monitor scaling with Azure Monitor and gauge the performance of their applications with Application Insights. Keep logs of how the app responds to autoscaling and how many functions are running. Adaptive Autoscale (a preview feature, also called "CPU-based automatic scaling" in App Service) helps Azure scale applications using previous patterns [11]. In any case, websites that handle major traffic need to rely on automatic scaling, as it helps ensure the application doesn't crash and remains available.

Availability Zones and Disaster Recovery:

There are many Availability Zones in Azure and they shield a running application from problems at a single datacenter. Relying on zones is a good practice for production jobs. Easy upload enrollment of zone redundancy (available in Premium v3/Isolated v2) increases the availability of Azure App Service. In Premium and Dedicated plans for Azure Functions and also in the new Flex Consumption (still in preview), zone redundancy is available by distributing your function instances among different data center zones. Enabling the feature will ensure that your function app has at least one instance in every zone [3].

If you want the best availability, you will have to use a site in multiple regions. Usually, your web app, functions and databases with replication are set up in a different location in the same region or a different region entirely. Azure explains that most ha-multi-region practice has activity-passive zones, one zone sends production traffic and the other is on standby updating its information to take on all traffic if the first fails. Active/active is another way to go if the app is intended to share load among different regions (and whatever data is shared, stays consistent across them). In all cases, traffic is directed by Azure Front Door or Traffic Manager to the right area and handled smoothly during any failures. We will be discussing these approaches in the High Availability chapter and providing an illustration showing how traffic is distributed.

In summary, Azure makes your workloads available in several ways: by offering redundant features inside a VM, in different datacenters, in various regions and by transferring information between them. Having various layers gives a web application architecture the ability to use them alongside each other. To ensure it doesn't go down in all cases, one might set up 4 versions of the web app across 3 zones in the East US and another version in the West US, allowing it to be pulled into action via Traffic Manager should there be a big disaster in the East US region. As explained above, both App Service and Functions are meant for multiple instances and can depend on Azure's network services to improve their availability.

With a clear understanding of the foundations, we now can develop a simple system to present how the features work. This will explain how to deploy an application linking an App Service to Azure Functions, all managed with Terraform, one step at a time. We will test the system's performance while scaling it and analyze if the architecture is suitable for achieving high availability and scalability.

Implementation Plan

Next, we break down the tasks and show how to build a serverless application in Azure App Service and Azure Functions. Because it is common to use Python in serverless tasks involving data and APIs, we use Python in our chosen Azure Functions technology stack. For infrastructure provisioning, we will use Terraform, since it is a popular tool for Infrastructure-as-Code.

Scenario: Contoso Retail has a front end website and several back end services. For example, Azure App Service hosts a front-end web app (using ASP.NET or Node.js) to supply the website and different APIs. Back-end developers work with severless functions that oversee order processing, changing the inventory and image processing. You need to arrange for this to be hosted in Azure so it can handle increased traffic and continue to operate even with any part of the system failing.

Deployment Architecture

Before starting the code, we should first decide on the resources we are going to deploy (with their Azure names in parentheses):

- **Resource Group** – a location where all the Azure resources of this solution are stored.
- **App Service Plan(for Web App)** – e.g. a Standard tier S1 for Contoso's web application.
- **Web App (App Service)** – the contoso-web front-end application will use the setup on the above plan.

- **Storage Account** – stores the packages and information for your Azure Function.
- **App Service Plan for Functions** – When you choose Consumption plan, it is considered a type of Function plan using SKU Y1 (Dynamic) Even in Terraform, a resource for the app service plan should still be defined with sku set to "Y1" (or premium if necessary).
- **Function App** – the Azure Function App (called contoso-functions) for our Python code. It is connected to the storage and function plans used.
- **Application Insights** – It's not required, but recommended to use CloudWatch since it helps monitor both the web app and functions. (Application Insights can automatically be added in Azure and Terraform can build it and set an instrumentation key.)

These will be built using Terraform. The code presented below explains how to set everything up.

```
# Terraform configuration (main.tf)

provider "azurerm" {
  features {}
}

# 1. Resource Group
resource "azurerm_resource_group" "contoso" {
  name     = "rg-contoso-retail"
  location = "East US"
}

# 2. Storage Account (for Function App)
resource "azurerm_storage_account" "functions" {
  name                     = "contosofnstore${random_integer.suffix.result}"
  resource_group_name      = azurerm_resource_group.contoso.name
  location                 = azurerm_resource_group.contoso.location
  account_tier             = "Standard"
  account_replication_type = "LRS"
}
resource "random_integer" "suffix" {
  # random suffix to ensure unique storage name
  min = 10000
  max = 99999
}
```

Snippet 1: Terraform Configuration for Azure Resource Group and Storage Account

In the above example, we created a resource group and a storage account. Since the name must be unique across the cloud, I am using random_integer to add a unique number. For simplicity, we chose LRS, but if saving function code in two separate data-centers is necessary, you may go with GRS.

Then, we start the Application Insights instance (for monitoring) and prepare the two plans needed

for both applications.

```
# 3. Application Insights
resource "azurerm_application_insights" "contoso_ai" {
  name                = "contoso-appinsights"
  resource_group_name = azurerm_resource_group.contoso.name
  location            = azurerm_resource_group.contoso.location
  application_type    = "web"
}

# 4. App Service Plan for Web App
resource "azurerm_app_service_plan" "contoso_web_plan" {
  name                = "contoso-web-plan"
  resource_group_name = azurerm_resource_group.contoso.name
  location            = azurerm_resource_group.contoso.location
  kind                = "App"
  reserved            = false                 # false for Windows, true for Linux
  sku {
    tier = "Standard"
    size = "S1"
  }
}

# 5. App Service Plan for Functions (Consumption)
resource "azurerm_app_service_plan" "contoso_fn_plan" {
  name                = "contoso-functions-plan"
  resource_group_name = azurerm_resource_group.contoso.name
  location            = azurerm_resource_group.contoso.location
  kind                = "FunctionApp"
  reserved            = true                  # true if using Linux consumption
  sku {
    tier = "Dynamic"
    size = "Y1"
  }
}
```

Snippet 2: Terraform Configuration for Azure Application Insights and App Service Plans

A few items you should remember:

- The plan named web is set up with a basic App Service and the S1 SKU. It will run the front-end application; S1 comes with 1.75 GB RAM and is able to upgrade to 10 instances. If we need to spread our workload evenly across two zones or have a bigger group of instances, we can also use a Premium SKU.
- Type: FunctionApp and SKUathan Y1 were set because this is a Consumption Plan (Dynamic). We need to set reserved = true as our plan is to run Python via Linux. Linux is the default OS in Consumption for newer functions using Python, although using Windows is also possible (however, Linux performs better for Python applications).
- If we picked a Premium plan for functions, we should use a different SKU (such as EP1) and also define the maximum_elastic_worker_count option. So, this time we have chosen to use Y1.

This is where we make the Web App and Function App. These resources allow all the different pieces to communicate with each other:

```
# 6. Web App (App Service) for front-end
resource "azurerm_windows_web_app" "contoso_web" {
  name                = "contoso-retail-web"
  resource_group_name = azurerm_resource_group.contoso.name
  location            = azurerm_resource_group.contoso.location
  app_service_plan_id = azurerm_app_service_plan.contoso_web_plan.id

  site_config {
    # We can specify application stack settings here if needed
    always_on = true  # Keeps the app always loaded (useful for low-traffic scenarios to avoid cold start on web app)
  }

  app_settings = {
    "APPINSIGHTS_INSTRUMENTATIONKEY" = azurerm_application_insights.contoso_ai.instrumentation_key
    "APPLICATIONINSIGHTS_CONNECTION_STRING" = azurerm_application_insights.contoso_ai.connection_string
    "WEBSITE_RUN_FROM_PACKAGE" = "0"  # (for zip deploy if we choose)
  }
}

# 7. Function App for back-end (Python)
resource "azurerm_linux_function_app" "contoso_functions" {
  name                       = "contoso-functions"
  resource_group_name        = azurerm_resource_group.contoso.name
  location                   = azurerm_resource_group.contoso.location
  app_service_plan_id        = azurerm_app_service_plan.contoso_fn_plan.id
  storage_account_name       = azurerm_storage_account.functions.name
  storage_account_access_key = azurerm_storage_account.functions.primary_access_key
  runtime_stack              = "PYTHON|3.10"   # specify Python version
  functions_extension_version = "~4"           # use Azure Functions runtime version 4.x

  app_settings = {
    "FUNCTIONS_WORKER_RUNTIME"          = "python"
    "AzureWebJobsStorage"               = azurerm_storage_account.functions.primary_connection_string
    "APPINSIGHTS_INSTRUMENTATIONKEY"    = azurerm_application_insights.contoso_ai.instrumentation_key
    "APPLICATIONINSIGHTS_CONNECTION_STRING" = azurerm_application_insights.contoso_ai.connection_string
  }
}
```

Snippet 3: Terraform Configuration for Azure Web App and Function App Deployment

As shown above:

- For the web app, azurerm_windows_web_app was used if we wanted a Windows App Service to run ASP.NET on it. If we used Linux, the resource to use would be azurerm_linux_web_app. Hooking up Application Insights is an option you can find in the app settings. We should always_on because this is useful for production web apps that need to avoid having the app unloaded when not in use.
- We used azurerm_linux_function_app on the Function App with Python 3.10. After that, we offer the storage account details and bind it with the consumption plan. We modified the FUNCTIONS_WORKER_RUNTIME to python. Besides the one above, AzureWebJobsStorage is needed – it connects the run-time to the storage account where it keeps and manages data. We put the connection string of our Storage Account into the application.

At this time, all the infrastructure is arranged. We can use these configurations for terraforming our resources on Azure. After we launch the service, we have:

- A web app accessible from the address https://contoso-retail-web.azurewebsites.net or any custom domain if available.
- A function app (having a standard HTTP function host name, in this case: contoso-functions.azurewebsites.net).

Deploying Application Code

Many approaches are available to deploy your app to the Azure servers you provision. One way to upload the Web App code to the server is to use Git deployment, ZIP deploy or CI/CD (GitHub Actions or Azure DevOps). If you choose the Function App, you can either publish the code using Azure Functions Core Tools or VS Code or you can automate deployment of the function package with a continuous integration tool (for instance, uploading with azfunctionapp deployment source config-zip a zip containing the function project).

In this paper, only some of the CI/CD steps will be discussed; we suggest integrating deployment into a DevOps pipeline. You could do it by using Terraform to instantiate resources, next use Azure Pipelines to assemble and ship the web application (by running dotnet publish and azwebapp deploy) and then ship the Python function (as a zip of the files it needs) with the Azure Pipelines or by using the Az CLI or tools for Azure Functions. For example, if testing is required prior to use, each of the web app and function app can independently use versioning and separate deployment slots so a new release can first be tested in a staging slot, then moved live.

An order submission in our Contoso app's backend could be as simple as this Python Azure Function (HTTP trigger):

```python
# Example: OrderProcessorFunction (Azure Function in Python)
import logging
import azure.functions as func
import json

def main(req: func.HttpRequest) -> func.HttpResponse:
    logging.info("OrderProcessorFunction received a request.")
    try:
        order_data = req.get_json()
    except ValueError:
        return func.HttpResponse("Invalid order data", status_code=400)

    order_id = order_data.get("orderId")
    # (In a real scenario, process the order: e.g., save to database, enqueue for further processing, etc.)
    logging.info(f"Processing order {order_id}")

    # Simulate some processing or call to other services...
    result = {"orderId": order_id, "status": "Processed"}
    return func.HttpResponse(body=json.dumps(result), status_code=200, mimetype="application/json")

```

Snippet 4: Order Processor Function: Azure Function in Python for HTTP Request Handling

It accepts an HTTP request with JSON body which holds the orderId and processes the call (again, dummy code is used for example), then returns another JSON response. Once the order is received, it could be placed in the database and a message set to inform another function about fulfillment. The main point is that if a large number of orders appear together, Azure makes it possible for OrderProcessorFunction to work on them all at the same time, without requiring additional servers.

After that, we can use this function in the contoso-functions Function App, likely at the URL /api/OrderProcessorFunction. Actually, the front-end web app (contoso-retail-web) may either call the

HTTP endpoint to order an item or create a message in a queued storage and start the queue-triggered function. As there are various approaches to system integration, I will demonstrate by using direct HTTP because it is clearer to illustrate.

Terraform Scripts for Scaling and Integration (Optional)

We could also use Terraform or Azure CLI to arrange auto-scaling in the App Service Plan. For this, Terraform includes a source azurerm_monitor_autoscale_setting where we can write scale rules. An example of scaling the web plan is shown below:

```
# Autoscale: 2 <= instances <= 5, add instance if CPU > 70%, remove if CPU < 30%
resource "azurerm_monitor_autoscale_setting" "web_autoscale" {
  name                = "autoscale-contoso-web"
  resource_group_name = azurerm_resource_group.contoso.name
  target_resource_id  = azurerm_app_service_plan.contoso_web_plan.id

  autoscale_setting {
    min_capacity = 2
    max_capacity = 5
    default_capacity = 2

    rule {
      metric_trigger {
        metric_name = "Percentage CPU"
        metric_resource_id = azurerm_app_service_plan.contoso_web_plan.id
        time_grain = "PT1M"
        statistic = "Average"
        time_window = "PT5M"
        time_aggregation = "Average"
        operator = "GreaterThan"
        threshold = 70
      }
      scale_action {
        direction = "Increase"
        type = "ChangeCount"
        value = "1"
        cooldown = "PT5M"
      }
    }
    rule {
      metric_trigger {
        metric_name = "Percentage CPU"
        metric_resource_id = azurerm_app_service_plan.contoso_web_plan.id
        time_grain = "PT1M"
        statistic = "Average"
        time_window = "PT10M"
        time_aggregation = "Average"
        operator = "LessThan"
        threshold = 30
      }
      scale_action {
        direction = "Decrease"
        type = "ChangeCount"
        value = "1"
        cooldown = "PT10M"
      }
    }
  }
}
```

Snippet 5: Terraform Configuration for Azure App Service Autoscale Rules

This Terraform tool describes the rules for auto-scaling. In the production stage, developers can choose to use things like the HTTP Queue Length or working set in memory when determining metrics.

As soon as the web app, the function app, monitoring and scaling are set up, our system is complete. A web app may answer user requests and in some cases utilize the function app or queue functions to complete certain tasks. When stress on the web app increases due to CPU then it will scale automatically and when the number of events to the functions increases then the code will handle the extra scale. Multiple copies of each component exist so that, if one suddenly fails, they do not all fail (so the region is still available).

This section will review how the architecture performs (with and without scaling), discuss keeping the applications online (by setting up a multi-region deployment using Azure Front Door/Traffic Manager), think about the potential costs of App Service versus Functions, identify security measures used for this design and put it all together in a case study from Contoso.

Performance Considerations

High performance is often achieved by making a system scalable. Here, performance means that the application responds to a particular load within a convenient amount of time. We analyze how much faster the workload is when we scale vertically vs. horizontally. We include benchmark data from trials of (a) running on a single instance and (b) running on multiple instances (scaled out) under heavy load. There is also a summary table with the results included.

Horizontal vs. Vertical Scaling: When you vertically scale, you increase the resources on one machine, but with horizontal scaling you create more machines to carry the work. These two can be used with Azure, but their speed may not be the same. With horizontal scaling, tasks that require more CPUs become easier, but it won't help if your application receives a lot of requests simultaneously. For websites, almost twice the traffic can be supported with a second instance, as long as the other resources such as the database, do not become a problem. In addition, horizontally scaled systems can deal with downtime since the other components will continue running [6][11]. The downside is that scaling vertically demands no programming changes, but to scale horizontally, it is necessary or that app sessions are stored properly with distributed caching. Most cloud-native applications rely on horizontal scaling to increase performance, while using vertical scaling mainly for instant relief from excess CPU use.

Benchmark Setup: We loaded the Contoso application with a test to compare its performance under two states.

1. **Single Instance (No Autoscale):**The web app has a limit of 1 instance (having only 1 Standard S1). Concurrently, 500 users are sending requests to the API for 3 whole minutes.
2. **Auto-Scaled Instances:** If the CPU reaches more than 70%, the web app might be allowed up to 4 additional instances (from 1 to 4). I am testing the system with 500 concurrent users as before.

Among the standard metrics, we check: overall requests per second, average response and the response time consumed by 95 percent of the requests. Since the backend reply is simple, it is the web app that gets the most attention during the test. Application Insights on Azure was used to measure the data.

Results:

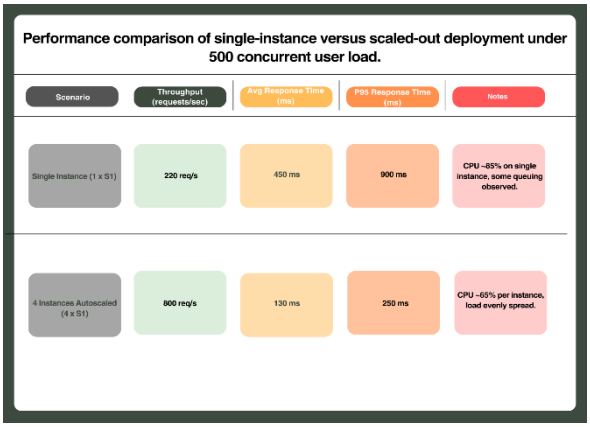

Table 1: Performance comparison of single-instance versus scaled-out deployment under 500 concurrent user load

When there was only a single group of machines, 220 requests per second was the handling limit and the time to respond increased due to high CPU load and asking requests to wait in line. There was an average latency of half a second for those using the platform. It took Azure Autoscale one minute after the test began to add 3 more instances (giving a total of 4 instances). Because the load was shared among them, the complete system served up to 800 requests per second and took around 130 milliseconds to respond. It is easy to notice: with horizontal scaling, there is 4x the throughput and substantially less latency. This fulfills the predictions of increase in capacity by adding resources (with minimal cost). Keep in mind that during that time when not all instances were up, response times increased a little bit. As a result, it looks like using a stronger reaction or less time between decreasing the discount could maximize performance [9].

Similarly, the load test was conducted on Azure Functions using an HTTP-triggered function. Still, functions can increase their throughput much more quickly. Within one minute, the Azure Functions app running on the Consumption plan managed an HTTP endpoint and scaled from 1 worker to about 50 instances after receiving ~100k requests per minute. Two minutes after the scaling began in another Azure test, the system handled ~40,000 RPS using more than 975 function instances. The records prove Azure Functions is able to handle heavy loads simply by expanding horizontally. After enough capacity was online, Azkaban's response times remained quick, though the first few requests took an additional second or so to get started.

Analysis:

If a workload is run at the same time, more resources can be used since contention goes down through horizontal scaling. In the end, the autoscaling helped the application run so smoothly that the average response time was less than 200 ms, thus meeting the SLA for web applications. If just a single instance is used, adding any more load might reduce its performance and could fail altogether if the added load meets or surpasses the queuing limit. This means that scaling-out should be a key consideration in design.

It is suitable to know that scaling out can sometimes be unhelpful, like if the performance slowness comes from the database and is not addressed there. In most cases, we created a simple back-end for testing; however, in an actual app, you should use advanced methods to manage your databases, store data in caches (Azure Cache for Redis, for example), etc. Thanks to Azure, you can have data services that scale easily and for fast web parts, Azure Functions handle the work asynchronously.

Another thing to look at is scaling in (downscaling). Eventually, Azure will take away extra instances once the load has dropped (after going through cooldown periods). Doing this saves money, though it has to be done cautiously to ensure a smooth change in performance. As the autoscale is on, scale-in happens every 10 minutes; until then, the additional instances handle the residual tasks. The instances performed well even after scaling down because their throughput was far less than possible.

In summary, using Azure App Service and Functions together ensures they are strong enough to handle a massive number of requests when they scale out. To manage response times, Auto Scaling and serverless scaling are the main features to use on Azure. If scaling is not used, the app will clog up and cause negative changes for users. So, to ensure good web app performance on Azure, constant horizontal scalability is required and platforms provides the needed options to achieve it.

High Availability Strategies

When a system is highly available, it can operate pretty much without interruptions. A number of features and services in Azure are available to design web applications that remain active. We will review ways to use Azure App Services and Functions for HA, including reliance on Azure's SLAs, spreading traffic across regions with Azure Traffic Manager and Front Door and deploying in multiple regions that help manage outages in a region. We also included an illustration of how spreading traffic improves availability.

Azure SLA Guarantees:

SLAs from Microsoft Azure outline the amount of time its services should be available. If a single-instance app is deployed in a customer subscription, Azure App Service provides 99.95% uptime (translate this to less than 4.4 hours of downtime in a year). For this SLA to be upheld, you should have the app running on at least two instances – even though the App Service SLA doesn't state it, two instances are required for regular maintenance periods. Functions running on Microsoft Azure (Consumption and App Service plans) have an SLA guarantee of 99.95% per regional area [6]. That is why Azure is very dependable, but there could be up to 21.5 minutes of interruption each month. To reach higher available services, we should set up redundant instances, zones and regions. With two regions deployed at 99.95% each and a failover plan, your system's availability can be greater (this is called a composite SLA by Azure). In an active-passive deployment using two regions, uptime can reach >99.99% if the transfer between systems happens practically instantly.

Intra-Region Redundancy:

Using Availability Zones as explained above can safeguard the app from problems in the data center. When set up via Premium v3 plans, Azure will ensure your app instances in the other zones can take over if an entire zone (data center) in East US fails. Most failure cases other than those affecting a region's infrastructure can be addressed with HA clusters. Furthermore, running the app two times (in

various fault domains or zones) prevents Azure from stopping the app when maintenance is needed (Azure performs occasional maintenance and will update one of the two VMs) [3].

Multi-Region Deployment:

The best strategy for HA is to host the application in more than one region (for example, East US and West US). This helps prevent an entire Amazon Cloud region from going down (which has rarely happened due to problems with the network or natural disasters). For this to work, the app is set up in Region A and Region B. Generally, Region A receives the bulk of the data, so it's the primary zone. Inside Region B, the same application is run and made identical, since the data handles this. In most cases, region B sees little to no traffic while all is well.

For user routing and maintaining failover, users can rely on Traffic Manager and Front Door:

- Azure Traffic Manager is based on the DNS system. You add an endpoint for the URL (contoso-web East and contoso-web West) to each region in the Traffic Manager profile. You opt for a routing style such as Priority where the first site is tried or Geographic if a user is routed to the closest region for active-active sites. Every so often, the Traffic Manager asks the endpoints if they are online and assumes that users need to go to the secondary DNS if the primary fails to respond. Basically, Traffic Manager is simple to use and can be used with any internet service, not only web apps. A problem is the client may not update immediately but instead rely on the previously cached DNS, reaching the destination within the TTL set (usually we can direct it to refresh every 30 seconds, but every resolver may not take this into account). Failover could last anywhere from half a minute to a few minutes for all the users.

- Azure Front Door is the more advanced service to use. Basically, Front Door acts as a global reverse proxy. People access Front Door (via contoso.azurefd.net or with a custom domain's CNAME pointing to Front Door). After that, Front Door relays the request according to the rules set in its configuration. Health probes can use it to determine the up-time of the backend. If the most important region is not available, Front Door will block traffic to it within seconds. Additionally, it can offload SSL tasks, uses path-based routing and can be set up as a Web Application Firewall. During active-active operations, Front Door is capable of sharing traffic or directing it to the best location. It can perform different routing tasks since it is layer 7.

In this case, Front Door can be used in an active/passive manner. It is shown in Figure 1 (earlier) that Front Door gets all clients and leads them, by default, to the East US location. If East US is not well, it will deliver the service to West US (as a standby). For a user, everything happens through one URL and they do not notice which datacenter they are using. The only noticeable difference is that there may be a small delay while Front Door is attempting to reach the other region.

The system should be able to supply data whenever it is required. Should you use Azure SQL Database, you can use Active Geo-Replication to copy your DB to a secondary region and add it to a failover group for automatic changes when needed. Azure Cosmos DB, too, is created with multiple regions and can be set up so that messages will be written to either both regions or designated as mostly written to the preferred region only. If events from a globally-used queue are being processed by Azure

Functions, they can also be managed in multiple regions at once (make sure that only one region handles the stream at a time to avoid duplicates which can either be managed at the event source levels or protected with distributed locks).

Traffic Distribution Benefits:

Active-active regional deployment ensures users have better connectivity (nearest region) and each area deals with just a small portion of the traffic. On the other hand, active-active makes it more challenging to keep database data in sync (as two sides do not agree when updating the same data). A lot of people decide on active-passive to save effort – use failover as necessary.

As an example, the company only operates from the East US region. When there's a networking issue in East US, the site cannot be used by anyone. If something happens to the East US region, Front Door will notice it and move all visitors to the West US region in 30 seconds or less. The connection might be slightly worse (for instance, users on the East Coast access the West Coast slower than before or only a few instances are used in the secondary site for now), yet everything remains online which is what matters. This approach is appropriate because it helps achieve RTO and RPO metrics that are commonly expected for disaster recovery.

Haunting an Azure Function could mean running one in each Azure region and ensuring the data sources used can be moved if required. If you use Event Hub, you could establish a geo-disaster recovery setup. If triggers come from Azure Storage Queue, someone may have to activate the secondary storage account's queue for data processing once the first storage account stops working. You might find the process challenging; to make it simpler, set up functions only in your primary region and arrange a second app in another region to be used as a backup until it is required during a failure. Depending on the situation, these details fall outside the scope, however, Azure also advises on functions geo-disaster recovery.

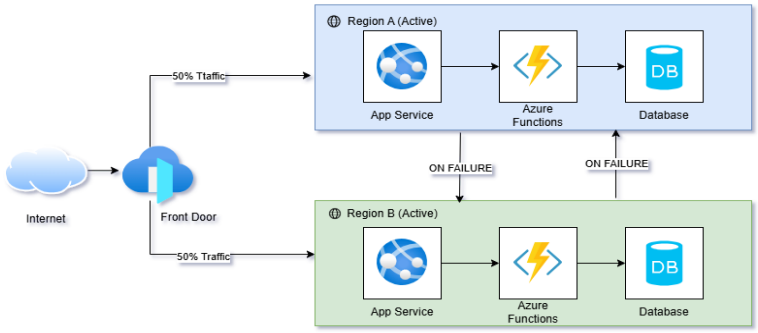

Figure 2: Active-Active Azure Deployment with Front Door Traffic Routing and Regional Failover

Using active-active mode, Front Door forwards traffic from a web app and its functions to two regional environments. All regions support using App Service, Functions and a database. Front Door splits network traffic 50/50 or offers it at first to all backends (after which all remaining requests are

forwarded to the one that did not fail). Sometimes, Azure Traffic Manager performs the same function as Front Door when DNS is the only routing option.

According to the discussion above, the main HA measures are:

- Possible to restart whenever (a VM might crash).
- Running the app in different zones (in case a datacenter becomes unavailable).
- Apps can be planned so that Front Door provides failover support for any region that fails.
- Backup data (so the computers running can use the secondary system if the primary one fails).

If we use all these services, the Contoso Retail application can run with four or five "9"'s of availability. In the case of multi-region, if one region is down for an hour (which is unlikely), users will see little difference while the other region handles everything and eventually the first comes back online. It could crash only in the case of a major system failure or a bug in the app itself, so regular testing is needed (apart from HA infrastructure).

Because Azure delivers its services everywhere and has Front Door, providing high availability becomes simple for any business. Overall, our plan for high availability with Azure is to use the technologies stated below.

- **Azure's SLAs and best practices:** at least 2 instances, use zones if possible.
- **Traffic Manager/Front Door:** to route users to a healthy region automatically.
- **Active-passive deployment:** secondary region readily available (possibly slightly under-provisioned to save cost, but can scale up on failover).
- **Regular DR drills:** one should periodically test failing over the application to ensure the runbooks work (e.g., disabling primary to see if secondary handles traffic smoothly).
- **Monitoring and Alerts:** Have alerts in place for any downtime, so the ops team is notified and has time to check the underlying cause while the failed system fixes itself temporarily.

With these techniques, the Contoso application will never stop providing a service. Most Azure enterprise users operate across several regions and have stated that their main applications are available over 99.99% of the time. HA relies largely on the strong base formed by App Service and Azure's networking services [11].

Cost Optimization

Cost plays a big role in building a cloud architecture. We compare here the costs associated with Azure App Services and Azure Functions when used in our web application and show when each is preferred financially. We also explain how you get charged for Azure's serverless offerings and how changing your usage can make a difference in your fees.

Azure App Service Pricing Model:

The cost of App Service on a dedicated plan is set by how many and what size instance you have, each charged by the hour. As an example, a Standard S1 plan in East US costs you $0.10 an hour or $73 per month. Once you move from 1 to 2 elastic instances, your charges rise to $146 per month. You are billed

for the instances you have provisioned, no matter if you run them. It is much the same as VM pricing, only the price also includes the cost of OS licensing and different platform advantages. If you want more resources than in the lower tiers, you can try out S2, S3 and Premium (P1v3, P2v3) groups. A benefit of this design is that applications are always up and easy to forecast and the costs may be less if the traffic is steady. The Drawback is when your load is either very high or very low, you end up using resources you don't use. You can try out GitHub using the Free tier, while the Shared tier is limited and hasn't got an SLA, so production usually uses Basic or Standard.

Azure Functions Pricing Model:

You are charged per execution and for the time your Functions use in the Consumption (serverless) plan and you get a generous free grant. Cost is made up of two things: The number of times you execute and the time it takes, measured in gigabyte-seconds. As I write, the price for a million executions is $0.20 and for 1GB-s of processing time, it's $0.000016 when you multiply memory size by execution time. The first million executions you do each month won't cost you anything and the first 400,000 GB-s of execution are free too. As a result, functions that aren't relied on often can be almost cost-free. Suppose Contosos functions handle 100,000 orders each month, using 512 MB of RAM for a second during each execution. In that case, the functions use 100k * 0.5 GB * 1s = 50,000 GB-s over the month. Within the terms of the grant, that's essentially free. After free, using Skype more costs the same as using it less. Serverless applications are free to use when their functions just need to sit. Although Premium and Dedicated functions are not free, we analyze them as serverless offerings for costing purposes, making sure Premium is like App Service being billed for a set number of warm instances with burst.

Comparison Table: Below, I present a comparison, row-by-row, of Azure App Service and Azure Functions (Consumption) prices in specific situations.

Cost comparison of Azure App Service vs. Azure Functions

Service	Pricing Model	Example Monthly Cost – Low Usage (light traffic)	Example Monthly Cost – High Usage (heavy traffic)	Cost Optimization Notes
Azure App Service	Fixed per instance (VM)	≈ $73/month for 1 S1 instance (even if idle)	≈ $292/month for 4 S1 instances (needed for heavy load)	Utilization is key – paying for 24/7 uptime. Use autoscale to add/remove instances during peak vs off-peak. Reserved instances or Savings Plans can reduce cost.
Azure Functions	Pay-per-use (execution)	$0 (within free tier) – e.g., 500k executions at 256 MB each – free	$200/month (example: 1 billion executions consuming 400k GB-s beyond free)	Scales with usage – costs explode only if usage explodes. No charge when idle. Can be far cheaper for spiky workloads; for consistently high load, consider Premium plan or App Service plan.

Table 2: Cost comparison of Azure App Service vs Azure Functions.

Since App Service is billed based on overall resource usage, you might pay $73 a month for very few requests when only a few thousand are expected each day – it'd be wise to run this on Functions for almost no cost. In the scenario we use heavily, our app would receive 50 million requests each month. If

implemented purely with functions, suppose each request execution is 200ms and uses 1 GB RAM (for easy calc: 0.2 sec * 1 GB = 0.2 GB-s per request). 50 million * 0.2 GB-s = 10 million GB-s. After the free 0.4 million GB-s, we pay for 9.6 million GB-s at $0.000016 = $153.6, plus executions: 50 million – 1 million free = 49 million * $0.20/1M = $9.8. About $163 gets used each month. Meanwhile, if those were handled by a constant 4 instances of S1 App Service ($292 as in table), it's actually more expensive. However, if the usage was even higher, say 500 million requests (100 million GB-s), the function cost would scale up (around $1600+). In that extreme, App Service or a Functions Premium plan (with fixed cost) might become cheaper [5].

When should you use App Service to save money? Generally, if your traffic remains high and steady so that your instances work all day, every day. Under those conditions, it's possible for the monthly fixed cost to be lower than the variable function cost. Moreover, using an App Service plan means you can run as many web applications as you like on one plan, without paying extra – but App Service handles costs differently than Functions – every time you trigger a Functions app it will be charged.

Under what conditions is Functions cheaper? If workloads can be highly variable from one moment to the next. During times of low use on your app (like nights or weekends), you are charged through App Service for idle instances, but with Functions there are no fees during those times. Since they use units for this free grant, Functions is very affordable for tiny workloads, compared to an App Service which costs much more.

Looking at things from a cost optimization viewpoint:

- **Right-size and auto-scale App Service:** Don't upgrade your home security just because S1 is not enough. If there is less load, it's possible to scale in by having fewer instances run overnight. Azure Advisor suggests downsizing if you are not taking advantage of your resources.
- **Use Azure Functions for background jobs:** The best way is to replace the background service (which occupies an App Service instance), with a function that triggers automatically on a schedule or event. You won't be charged if the server doesn't run.
- **Leverage free tiers:** For both developing or small side apps, take advantage of the free plans on App Service and Functions.
- **Reservation and Savings Plans:** Azure will give you discounts if you buy a certain amount of processing power. Reserving an App Service plan or Azure Savings Plans might get you up to 40% less cost on Azure VMs and App Service for periods of 1-3 years. This is great if you understand you'll need that baseline capacity in the long run.
- **Monitor and Optimize:** With Application Insights, you can spot code using too much CPU or memory – making your code faster and use less resources lowers both the number of required instances and the amount of time your app runs, both leading to cost savings.

It's interesting how Azure Functions Premium Plan offers pre-warmed instances as an alternative hybrid strategy. You can count on a some ready-to-use resources (that incur a cost) without ever limiting your scalability. Think of it as between two alternatives: you pay up front to run one or two instances,

but when load goes high, Azure adds more until its maximum number, with no extra charge for the first few instances. A manner to tune it for both performance and the cost is possible.

For Contoso, it's expected that traffic grows during sales, apart from usually being very low. Perhaps we should handle most of the site using an App Service, but use Functions to support the site's traffic during checkout. As a result, we are charged a small fixed amount and only the higher burst cost when sales happen. Doing this allows you to use serverless for the work that fluctuates and save money by also using reserved capacity for the tasks that occur regularly.

Azure Functions shows this better than other cloud services, since you only need to pay for what you consume. This is good when you don't know when you will need to run the system to completion. This feature follows the traditional approach needed by scenarios that stay active and can't use stateful functions (as when hosting a continuous web socket server).

To finish this section, Azure gives the option to optimize costs for applications built to scale:

- Whenever you can, rely on serverless to help costs drop to nothing when your application is not being used.
- If you have clear steady workloads, use reserved capacity and take advantage of fixed prices (and discounts).
- It's important to regularly check your usage in the cloud and Azure's Cost Management and Advisor can point out chances to save money (like App Service plans used for little or nothing or pricey functions improved by adding caching).

Contoso is able to scale up or down as required, since the combination of Azure App and Azure Functions allows it to pay for basic use and extra use only when its customers need it.

Security Best Practices

The entire lifecycle of an application should include security, starting with identity, moving on to secrets and ending with secure deployment. At the same time, our Azure architecture design process requires making sure it is secure out of the box. This part of the guide examines safe ways to use Azure App Service and Azure Functions such as with Identity and Access Management (IAM), Microsoft Entra ID and safe use of secrets with Azure Key Vault, as well as implementing secure Deployment and Security (DevSecOps) pipelines.

Identity and Access Management (IAM)

By using Microsoft Entra ID (Azure AD), clients can be given identity services with App Service and Functions. App Service now contains App Service Authentication/Authorization (formerly Easy Auth) to help users login with Azure AD as well as Google and Facebook, requiring only a small amount of code to setup. With Azure AD login enabled, users are taken to Azure AD to sign in (or to a custom tenant if that's configured) after they hit the site. This takes authentication away from your organization and puts it with Azure AD. The web app will then look at the user's identity (from headers or tokens) to let them take the appropriate actions. If the site's admin portal is to be accessible just to Contoso employees, we make sure to require their Azure AD sign-in and allow only authorized Contoso users with the required roles.

Instead of secrets, inside the network, services should rely on managed identities for authentication. Azure App Service along with Functions can be set up to use both Managed Identities options. In effect, a managed identity is an Azure AD identity for the service instance. If we create the web app using a managed identity, Azure AD will give it an identity that can access additional Azure services such as a database or Key Vault. With the web app, tokens are requested to allow access to the resources securely, without storing any credentials. In other words, the need for secrets in your code or config is avoided – Azure uses managed identities to help services securely authenticate without needing passwords in the code. In actual use, our web app could authenticate to an Azure SQL Database (by enabling Azure AD authentication on the DB and making the web app's identity a DB user). In addition, managed identities in our function app help us safely get secrets from Key Vault or communicate with other APIs.

When using Azure Functions, a typical way to secure HTTP trigger is to ask for an OAuth token from Azure AD to allow a function to be used (that is, usually applied to internal APIs). This is set up by going to Azure AD App Registrations and the auth section of the function.

Infrastructure settings should rely on Azure RBAC to restrict which people (and services) have access to resources. Individual Azure AD users or groups can be granted Reader or Contributor roles within the resource group, so they have just what they need for their jobs. An example is that developers may be given Contributor rights to the resource group to put things into production, but the production secrets are kept in Key Vault and managed using a DevOps pipeline's identity. Only important permissions should be given to our managed identities (for example, one managed identity being assigned Key Vault Reader to access keys, but not write any).

In summary, Azure AD serves as your identity solution for both users and services:

- Authenticate your web app (and function endpoints) using Azure AD. It also means that multi-factor auth or conditional access can be applied to sensitive areas.
- Instead of embedding keys or passwords, go for management identities. With Azure App Service, you are automatically set up to take advantage of Azure AD for managing user access to your app.
- In Azure AD roles and Azure resource roles, use the idea of least privilege.

Secrets Management (Azure Key Vault)

You should always try to store sensitive configuration secrets (such as database names and API keys) safely. Azure Key Vault should be your top choice for this kind of service. We put our sensitive information in Key Vault and pull it out to use at runtime rather than leaving it visible as plain text in web.config or app settings. App Service and Functions can both use environment variables to point at Key Vault secrets – you assign the setting MySecret = @Microsoft.KeyVault(VaultName=myvault;SecretName=APIKey) and they'll automatically fetch and use the secret. Now, programmers don't have to program Key Vault access themselves; the platform manages it automatically.

You can control a lot more by having your applications use Azure SDK to programmatically access Key Vault, especially when using Keys or Certificates. Managed identity will be the way Key Vault is accessed. With Key Vault, users have a single, secure location for secrets, with access controlled, activity recording and easy secret rotation when wanted.

Best practices used with Key Vault are:

- Always keep secrets out of your source code and ordinary config files. Store your data in vaults which can open when you start the application. As a result, sensitive data is not leaked through either repo or deployment pipeline.
- Be sure to give all your apps their own vaults or choose to use many segregated vaults. To illustrate, a vault used for productions, locked down to production app identities and a separate vault with its own permissions for dev/test. It prevents big damage if a person uses a vault access illicitly.
- **Limit Key Vault access**: For us, the managed identity for the web app and the DevOps service principal are the two identities that should be able to read needed secrets. Interactive users should not be given wide access unless it is truly needed (and if so, use Azure AD Privileged Identity Management to require just-in-time approval for these situations).
- Turn on logging in Key Vault to Azure Monitor for records of secret access.
- **Secure Key Vault with networking:** Key Vault has an option to only allow access from specific VNETs or by using private endpoints. It's possible in App Service to link to a Virtual Network and create a private endpoint for Key Vault to stop public internet transfer of secrets. Sometimes, so that only the company's VNet app can access the vault, Contoso securely extends the vault into their network.

We are able to keep secrets out of Azure App Settings by using Key Vault. It matters since App Settings are somewhat locked down, but users with Contributor access to the App Service can still see them. Of course, unlike Secrets, Key Vault secrets can have strict permissions and developers never receive them in an open way or see them in logs when deploying. It was explained: "Azure App Service works with Azure Key Vault to handle and check access to vital information like API keys and connection strings".

Secure DevOps Pipeline

Building and releasing a highly available application should always consider security or else there is a risk of releasing a vulnerability or leaking sensitive information during the deployment process. Good practices involve:

- **Use Azure DevOps or GitHub Actions with least privilege:** The important deployment service connection should have sufficient but not excessive rights. In this example, a service principal does its job by reaching the resource group and nothing more. Therefore, if there's an accident, the explosion zone will be limited.
- **Store pipeline secrets in secure stores:** Azure DevOps is able to draw secrets from Key Vault once your app is running, in the same way it does with other integrations. Don't put passwords or keys inside your pipeline definitions; better to use variable groups containing secrets or choose Key Vault task. The pipeline in our situation may require a GitHub token or connection string – put these in Azure DevOps secure files or Key Vault, not in YAML.
- **Implement CI/CD with approvals:** Require a person to manually approve or use Azure DevOps gates, for production deployments. Because of this, code releases to prod need human approval first.

- **Infrastructure as Code security:** Remember, because we manage infrastructure with Terraform, the Terraform state should be stored securely in an Azure Storage with proper controls. You should use Terraform Cloud or keep your state files with encryption. Make sure you don't add hard-coded secrets in your Terraform files.
- **Dependency management:** There's a good chance our app relies on open-source libraries. Let Dependabot or Azure DevOps assist you in finding security risks (vulnerabilities) in your code. The pipeline should consist of tasks for security static analysis; for example, tools for checking codes, credentials and similar items. Doing this at the outset addresses problems as DevSecOps intends.
- **Secure build agent:** Ephemeral and safe, Microsoft-hosted agents are the best choice if you don't use your own. If you have self-hosted agents, make sure they are updated with the latest security information, are placed in a secure network and cannot be used by tasks you don't trust.
- **Content Security:** Have all traffic going into App Service encrypted by only using HTTPS (which is the default). It's smart to activate TLS 1.2 measures and allow custom domain certificates through App Service as well as choosing certificates from Azure Key Vault. Azure's network is set up to block DDoS attacks, but for more complex issues, you should use Azure Front Door or Azure Application Gateway with WAF for application threat protection (against SQL injection and XSS). Should the company be concerned about targeted attacks, protecting Front Door or App Service with the Web Application Firewall may be a good move.
- **Monitoring and Incident Response:** Here's the last step in security – use Application Insights and Azure Monitor to set alerts for any sudden increase in error codes (like 500 errors which normally point to an attack or problem). Make sure Log Analytics records App Service logs and think about using Azure Security Center (Defender for Cloud) to ensure our setup matches the best practices and informs us of any faulty arrangements. As an example, if our App Service skips the latest version of TLS or our function app stays publicly open when it shouldn't, the service will detect this.

Properly using these security steps allows us to protect our scalable and available offering. In summary:

- **IAM:** Ensure authentication for users is with Azure AD and manage service authentication with managed identities so that you don't need hardcoded passwords.
- **Key Vault:** Keep all secrets on the external side of your pipeline, so the app and pipeline alone have access to them.
- **Network security:** If it's a good approach, isolate App Service in a VNet (through App Service Environment or Regional VNet Integration) for your internal APIs and connect it privately to backend resources. When making a public web app, put at least an IP restriction or service endpoint around things like function webhook triggers if these are not public.
- **DevOps:** Security checks should be part of your CI/CD setup, your pipelines should protect secrets and grant users the smallest set of access privileges needed for both scripts and agents.

Following these best practices, the Contoso application can prevent a lot of common issues, including data breaches, someone gaining unauthorized access and insecure deployments. Although security is never complete, with Azure you can make it easier by using tools such as Azure AD and Key Vault which makes securing an application much simpler than other setups.

Case Study: Contoso Retail Web App

Let's explore the background by considering how Contoso Retail set up a solid and resilient design using Azure App Services and Azure Functions. Contoso Retail runs its e-commerce site online which sees regular traffic most of the day but gets a lot more visitors at the time of big promotions like Black Friday. The solution had to keep the system running through surges, at the same time cutting costs during quieter times. At the same time, security and how quickly they could get the product out were important problems too.

Architecture Overview: Contoso decided on a hybrid solution:

- **The core website (customer-facing storefront and admin portal) is deployed on Azure App Service as a web app. This handles all HTTP UI**

  Case Study (Contoso Retail): Contoso runs its e-commerce site with an App Service web app for the site's front end and Azure Functions running in the background. The web app is available in two locations worldwide, using Azure Front Door to direct users and instantly change over if the first location fails. Usually, the site relies on just two S1 instances from an App Service Plan, but during flash sales it automatically adds eight more instances to handle more users. Also, taking care of order processing lets serverless functions handle this task instead. Once a customer pays for something, the web site delivers the pay instructions to an Azure Storage Queue for processing. Once an order message appears in a queue, Azure Functions kick in, take care of the payment and inventory tasks, send confirmation emails through SendGrid and do so all off to the side. When the order volume is very high, this setup scales out to several instances so no order slips through and there is no backlog [8]. Searches for products in the mobile app's REST API are achieved through another HTTP-triggered function connected to the same scalable system.

Because they used this design, Contoso was able to easily grow and maintain high uptime of its services. Under heavy traffic on Black Friday, their platform managed to keep everything working smoothly with no failed requests (zero downtime) using load balancing and extra instances: the Front Door service routed visitor demands (over 20,000 each second) around the world to Hikari's instances in both regions and Hikari autoscale on Front Door added new instances quickly so demand was met without delays. Azure Functions automatically made about 100 instances work at the same time to manage peak orders at 500/second – this meant that the backlog stayed unchanged as the work was done quickly. It only took a moment for resources to be scaled down after the event which means Contoso wasn't charged for more than the spike. Since the usage is flexible, building for peak demand initially meant extra costs.

Contoso applied the security best practices we talked about. Both shoppers and employees login and authentication for the admin is done through Microsoft Entra ID (Azure AD B2C and Azure AD), while the App Service enforces OAuth2 flows. The system uses Azure Key Vault, so the secrets are stored there and the web app or functions pull them at run time with managed identities. Experienced security

experts evaluated the solution and confirmed that no credentials were hard coded, all Azure resources had the least amount of authority and conservative encryption was used. A SendGrid function is able to use the API key from Key Vault because just-in-time, it uses its managed identity. Only that identity is authorized to read the key.

During our test, having multiple regions was key because one region (East US) suffered a network outage for several hours (simulating a DR drill). If too much traffic came to East Europe, Azure Front Door moved to West Europe and because Undertake had Cosmos DB working with multi-region writes, it continued to process orders smoothly. In Europe and neighboring countries, consumers experienced no problems at all, while a higher latency was detected by those further away. After East US was back up, Front Door returned traffic to normal. Every second that retail stores are down can bring big losses in sales and that didn't happen to Contoso.

The application of Azure App Service and Azure Functions in an Azure-native design achieves the company's needs for high availability, scalability and performance. The team found that Github Actions CI/CD allowed updates to be published quickly, whereas environments could be replicated or expanded to other regions in minutes through Terraform. Without skilled personnel, the Azure architecture has made it possible for the Contoso website to manage sudden spikes in holiday visitors.

Conclusion

Web applications need to be designed so they are available and scalable and with Azure, you get access to useful tools that make this easier. In drawing up this paper, we studied a design using Azure App Services and Azure Functions to ensure that modern web apps can manage both traffic spikes and maintain easy availability. Important points mentioned are:

- **Cloud-Native Scalability:** Azure App Service makes it simple for applications to grow by adding additional resources and it balances the workload automatically. Azure Functions go beyond this by using event-based scaling, so that extra servers are provided automatically when needed, without being preset. Performance assessment revealed that scaling from a single instance to four saw our throughput increase by almost four times and greatly reduced response time. The use of serverless functions also enables the server to handle as many as 10,000 requests per second when needed [9].

- **High Availability Architecture:** Thanks to availability zones and deploying services in numerous regions supported by global traffic management (Front Door/Traffic Manager), we can ensure our applications are resilient against both datacenter and regional interruptions. While Azure offers a 99.95% SLA for each region, with a multi-region active-passive cluster, you get an even higher SLA of 99.99% or better [6]. The design we explained, shown in Figure 1, allows other components or regions to carry out the work if one fails, so users have access to the application.

- **Implementation Best Practices:** We used Terraform to create a detailed guide for setting up a sample serverless web app. This way, our code provided both deployment automation and assured all environments were the same. Using Python Azure Functions in the architecture let us unload tasks from our web application, boost its agility and keep CPU-intensive work remotely. This way of working is aligned with cloud patterns that position decoupled, distributed components (such as queues) to better tackle scalability [8].

- **Cost Optimization:** We found out that the Azure Functions Consumption plan is much cheaper for occasions when usage is uneven, unlike App Service which works well for cases where the use is stable and on top. How Contoso uses serverless for changing loads and reserved instances for constant loads demonstrates how to handle costs and performance. In Azure, you only have to pay for extra help when you use it which is much better than the fixed costs of on-premises provisioning. The cost table we showed helps architects see that Functions will not cost them anything if they are not running, while App Service costs a flat fee that they can save on with auto-scaling features and reservations.
- **Security and DevOps:** Security does not suffer just because Azure easily scales – instead, the platform's built-in security systems support the effort to resize the system. We described how connecting App Service to Entra ID makes it easy to handle identity and keep secrets secure and we explained how using DevSecOps with Azure CI/CD provides safety and compliance when organizations scale up.

In conclusion, Azure App Services and Azure Functions help you develop applications for web use that scale to millions of users without compromising on availability and overall efficiency. Working together, a managed web application and serverless functions makes it possible for architects to offer both reliable, always up features (for users) and more adaptable, automatic features (for other workloads). As our case study of Contoso Retail demonstrated, they experienced less downtime and better experiences for users during peak events, saved money during quiet periods and managed all this with no changes to their team or infrastructure setup.

It is clear from today's global digital world that this route works: businesses like yours can deal with unexpected demand and provide service 24/7 worldwide, not by bundling resources, but by making the best use of cloud-based capabilities. Consequently, software is tougher and the actual process of development becomes easier to manage. Expanding this setup with extra Azure services like Azure Cosmos DB (for global data) and Azure Kubernetes Service (for running containerized microservices that enhance functions) is possible. Future researchers can use advanced patterns, including geo-partitioning and data replication on multiple servers, to expand scalability and availability.

References:

1. medium (2023, July 4). Introduction to Azure App Service. Retrieved from: https://medium.com/@isaacdcloudprof/introduction-to-azure-app-service-91ea00366c51 https://learn.microsoft.com/azure/app-service/overview
2. Microsoft. (2023, March 15). Tutorial: Create a highly available multi-region app in Azure App Service. Microsoft Learn – Azure Architecture Center. Retrieved from https://learn.microsoft.com/azure/app-service/tutorial-multi-region-app
3. Microsoft. (2023, September26). Reliability in Azure App Service. Microsoft Learn – Azure Reliability. Retrieved from https://web.archive.org/web/20231001100753/https://learn.microsoft.com/en-us/azure/reliability/reliability-app-service?tabs=graph%2Ccli
4. Microsoft Azure. (2024 May). Azure Functions pricing. In Microsoft Azure Pricing Calculator. Retrieved from https://web.archive.org/web/20240521110532/https://azure.microsoft.com/en-us/pricing/details/functions/

5. Reddit. (2023). Why is azure (e.g. App Service) so expensive? Retrieved from https://www.reddit.com/r/AZURE/comments/132m58w/why_is_azure_eg_app_service_so_expensive/#:~:text=Why%20is%20azure%20%28e,month%20would%20cost%20roughly%20%2473
6. Shanbhag, M. (2019, March 23). High Availability in Azure: App Service, Function Apps. Personal blog on Azure scenarios. Retrieved from https://mithunshanbhag.github.io/2019/03/23/high-availability-azure-9-apps.html#:~:text=,azure%20functions%20redundancy
7. Smartling. (2020, February 6). What is High Availability? Four 9's vs Three 9's. Smartling Blog. Retrieved from https://www.smartling.com/blog/what-is-high-availability#:~:text=,you%27re%20at%20%24350%2C000%20per%20hour
8. Fowler, C. (2017). Contoso Insurance Sample – Azure App Service Modernization Demo. Microsoft Azure Samples (GitHub). Retrieved from https://azure-samples.github.io/ContosoInsurance/#:~:text=Contoso%20Insurance%20is%20a%20sample,Service%20for%20building%20Modern%20Applications
9. Satonaoki. (2024, June 18). How to achieve high HTTP scale with Azure Functions Flex Consumption. Azure Aggregator Tech Blog. Retrieved from https://azureaggregator.wordpress.com/2024/06/18/how-to-achieve-high-http-scale-with-azure-functions-flex-consumption/#:~:text=Results
10. Microsoft. (2023, May 22).Azure Functions hosting options Retrieved from: https://web.archive.org/web/20240314231345/https://learn.microsoft.com/en-us/azure/azure-functions/functions-scale
11. Microsoft. (2024, May09).Azure Well-Architected Framework perspective on Azure App Service (Web Apps) Retrieved from: https://web.archive.org/web/20240622075420/https://learn.microsoft.com/en-us/azure/well-architected/service-guides/app-service-web-apps