Azure container Apps: The future of Microservices in Azure?

Looking at the current state of software development, we can conclude a few things:
1. Containers are here to stay. Over the years, containerized workloads have become more and more popular, and we see most mature software companies benefit using containers from the cloud to the edge.
2. The DevOps movement is still growing and growing; the mantra "You build it, you run it" really works for building better software. DevOps teams must take into account the whole picture of building applications, from features to costs, from application monitoring and underlying infrastructure instead of only being responsible for building features for their applications.

Authors Geert van der Cruijsen and Bas van de Sande

Combining these two trends in the market explains why technologies such as Serverless became popular. Development teams must focus on everything related to building functional, resilient, and robust applications while taking costs into account. Serverless helps in reducing the amount of moving parts you must manage as a development team.

Kubernetes is another technology that took the world by storm over the past several years. Containerized workloads are popular, and Kubernetes gives you a vast number of options to deploy and run these workloads, either in the cloud or the edge, with flexibility between all clouds and self-hosted options.

Kubernetes also offers great tools for autoscaling, recovery of failing containers, zero downtime deployments, and controlling the network within the applications with service meshes. Because of that, all cloud providers have invested heavily to create ways to run Kubernetes on their clouds. That is why Kubernetes is becoming the standard infrastructure for modern cloud native applications.

There are also some downsides to Kubernetes. Managing Kubernetes itself is quite complex and although the public cloud providers are all investing in making running applications easier and easier, Kubernetes itself is still far from a PaaS or serverless service that needs little to no configuration for production workloads. In a world where we want to have T-shaped development teams that can build and run their applications, also having those teams know everything about Kubernetes can be quite a burden.

Microsoft acknowledged this and realized most companies do not need all the features Kubernetes has to offer. Their aim when building Azure Container Apps was to create an opinionated way of deploying containerized workloads to Azure that brings several features that Kubernetes could

provide without having to manage a cluster: autoscaling, zero downtime deployments and traffic shaping with control over ingress.

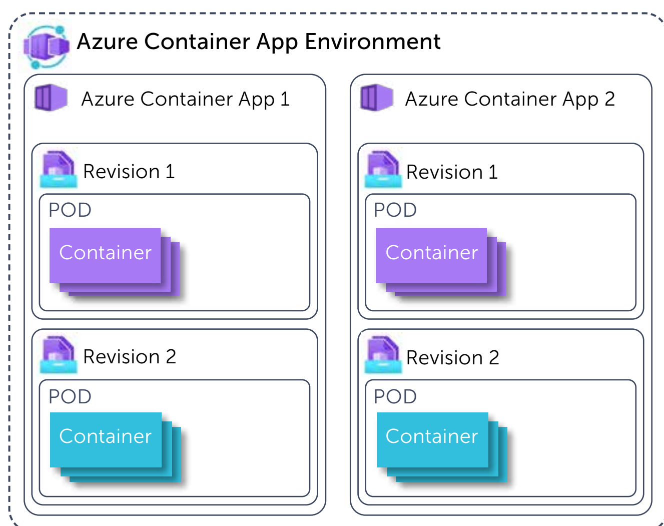
Introducing Azure Container Apps (ACA)

As mentioned, Microsoft aimed for creating an optimized way of deploying and running containerized workloads when building Azure Container Apps. Their focus was to build a solution that makes it easier for development teams to build Microservice architecture-based applications and deploy those to Azure. The idea being, giving development teams the features they really want from Kubernetes without having to deal with Kubernetes itself.

Azure Container Apps behind the scenes is still based on Kubernetes but as a developer you should not care about it. It has been set up for you, so it feels (and costs) like a serverless way of deploying containerized workloads to Azure, focusing on Cloud native applications and Microservice architectures.

What features does Azure Container Apps have to offer?

What are the features that development teams want when building and hosting microservices? ACA offers a way to deploy and scale a set of containers that make up an application while making sure all components can communicate with each other, scale based on load, are accessible from the outside, and can be deployed without downtime.



Looking at the components that define an Azure Container Apps solution, you always start with an "Azure Container Apps Environment". The Environment is a secure boundary around several "Container Apps" and makes it possible for these different container apps to communicate, much like an App Service Environment when using Azure App Services.

Within the Azure Container App Environment, you can create Azure Container Apps. Each app represents a single deployable unit which can contain one or more related containers. You could compare an Azure Container App to a deployment in Kubernetes. For each app, you can create a number of "revisions". Revisions are a way to deploy multiple versions of an app where you have the option to send the traffic to certain versions. Between revisions the ACA can be composed totally different: think of using different images, having additional containers etc.

These features are the basic concepts to run API's and frontends, but ACA also has features to host workers or background processes that are part of the microservice application. ACA has Kubernetes Event-driven Autoscaling (KEDA) built in. KEDA can scale background workers based on scaling rules , such as number of requests or the number of messages in a queue. These rules can be set up for each Container App individually, allowing them to scale based on their own needs.

Deploying an application to Azure Container apps

Azure Container Apps are built upon Kubernetes technologies, technologies which are hidden beneath the surface while deploying a new Container App. To get a better understanding of the technologies involved and the heavy lifting that is done, here is what actually happens when a Container App is deployed.

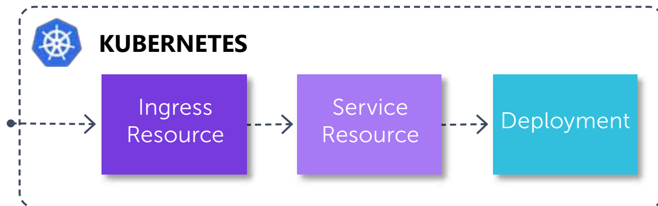
Containers can be deployed in Kubernetes in multiple ways. One way to rollout containers is by using deployments. A Kubernetes deployment can be defined as a yaml declaration describing which containers, storage volumes, and ports should be created, as well as the number of replicas. An example of a Kubernetes deployment which deploys three instances of the Nginx web server is shown below:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
   app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

Each time the definition of the yaml changes and is reapplied on the cluster, a new revision is made and the running deployment is updated gradually to the new revision. One of the advantages is the revision history is stored within Kubernetes, allowing the administrator to roll back the deployment to a previous revision.

When an Azure Containerized App is deployed to Azure, the app will be packaged as a Kubernetes deployment, leveraging the benefits of a Kubernetes deployment. Each update to the ACA will result in a new revision that can be rolled back if needed.

For the ACA to allow ingress, a Service and an Ingress resource are created as well in the underlaying Kubernetes cluster.



The Service resource is a static endpoint inside the cluster and a mapping for Kubernetes to tie containers to specific ports. This is done using the key/value pair in the selector. In the example this is "app: nginx".

```
kind: Service
apiVersion: v1
metadata:
  name: nginx-service
spec:
  selector:
    app: nginx
  ports:
  - port: 80
```

The Ingress resource then describes how the incoming traffic to the cluster is routed to the correct containers. In the example, when incoming http traffic on port 80 is detected, the traffic is forwarded to the service with the service name "nginx-service". The nginx-service will then route the traffic to all pods or deployments with the selector "ap: nginx".

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: nginx-ingress
  annotations:    ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - http:
    paths:
    - path: /
      backend:
        serviceName: nginx-service
        servicePort: 80
```

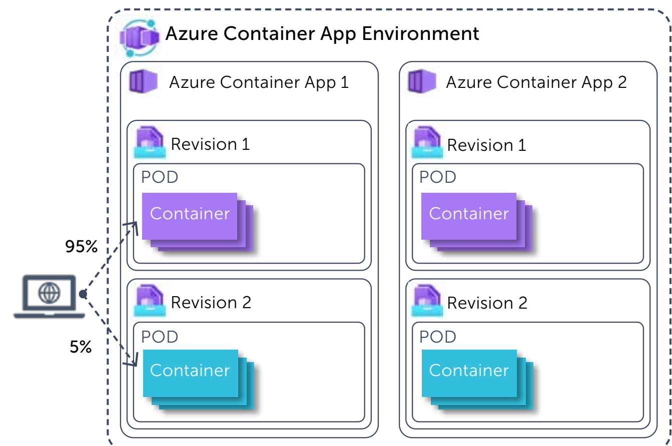
The equivalent of all this heavy lifting is done behind the screens when the following Bicep containerApp resource is deployed to Azure.

```
resource containerApp 'Microsoft.Web/
containerapps@2021-03-01' = {
     name: nginx-example
     kind: 'containerapps'
     location: 'west-europe'
     properties: {
     kubeEnvironmentId:  'xxxxxxxx'
     configuration: {
        secrets: secrets
        registries: []
        ingress: {
          'external':true
          'targetPort':80
        }
     }
     template: {
       containers: [
         {
            'name':'nginx'
            'image':'nginx:1.14.2'
            'command':[]
            'resources':{
               'cpu':'.25'
               'memory':'.5Gi'
            }
         }
       ]
     }
}
```

The template section in the Bicep example describes the containers to be deployed (`name: nginx`). The configuration section describes the equivalent of the Kubernetes service (`targetport: 80`) and the Kubernetes ingress (`external: true`).

Traffic splitting between revisions

Revisions in Container Apps allow us to split traffic between revisions to roll out new functionality gradually to our users.



Traffic splitting is done by adding traffic rules, in which the different revisions of the Container App get a different weight (note: the sum of the weights must equal 100).

```
{
  ...
  "configuration": {
    "activeRevisionsMode": "multiple"
    "ingress": {
      "traffic": [
        {
          "revisionName": <NAME_OF_REVISION_1>,
          "weight": 95
        },
        {
          "revisionName": <NAME_OF_REVISION_2>,
          "weight": 5
        }
      ]
    }
  }
}
```

In order to use traffic splitting, the `activeRevisionsMode` of the ContainerApp should be set to "`multiple`". If this mode is set to "`single`", a new revision would cause other revisions to be deactivated automatically.

Background workers in Azure Container Apps

Azure Container Apps bring the possibility to deploy "background" applications on Azure. These are applications that do not expose public endpoints and which can run forever.

By using standard Kubernetes Event-driven Autoscaling (KEDA) technologies, Azure Container Apps can scale up and down based on the number of events needing to be processed. The maximum number of replicas is set at 25 replicas. In most cases, Azure Container Apps can scale back to 0 replicas when they are idle.

Many KEDA scalers are available for a wide range of technologies (such as AWS, GCP, Azure, Redis, etc). Per Container App, the scaling metrics can be specified based on a number of rules that are different for each technology.

In the example below, the container app will scale up gradually to a maximum of 5 Replicas when the number of concurrent Http Requests is 100.

```
{
  ...
  "resources": {
    ...
    "properties": {
      ...
      "template": {
        ...
        "scale": {
          "minReplicas": 0,
          "maxReplicas": 5,
          "rules": [{
            "name": "http-rule",
            "http": {
              "metadata": {
                  "concurrentRequests": "100"
              }
            }
          }]
        }
      }
    }
  }
}
```

When you start working with Container Apps and KEDA triggers, documentation on the trigger specification can be found on the KEDA website (https://keda.sh).

The KEDA documentation shows code examples in YAML, while the Container Apps ARM template is in JSON. As you transform examples from KEDA for your needs, make sure to switch property names from kebab casing (everything in lowercase, with dashes between words) to camel casing (everything lowercase, all words after the first word start with uppercase).

Microservices using Dapr in Azure Container Apps

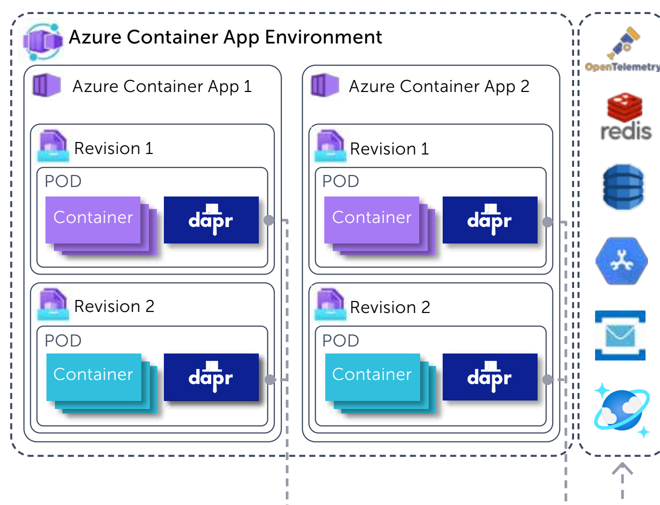
To make different components in the microservice landscape work together, ACA offers Dapr support out-of-the-box, by setting a configuration value to true. From there, on each app can use all the features of Dapr such as service location, pub/sub messaging, or distributed tracing.

Dapr is an open source project started a few years ago in the CTO office of Azure and in 2021 was donated to the CNCF Foundation and is now a CNCF Incubation project[1]. Dapr stands for "Distributed APplication Runtime" and helps developers focus more on their applications instead of knowing everything about the network, storage, monitoring tools, etc.

Dapr creates an abstraction for developers, so they only need one set of APIs to call other services, store state, or send messages. We wrote an article about Dapr in XPRT Magazine #10[2] and there are loads of information on the Dapr.io website.

Dapr works through a sidecar architecture. Azure Container apps make it possible by just checking a box to enable these sidecar containers to your Azure Container app without needing to setup anything yourself. By enabling this feature for your app, you can immediately use all the features Dapr provides.

Adding this feature again proves that Azure Container Apps chose an opinionated way of building containerized Microservice applications, although the use of Dapr is completely optional.



Azure Container Apps compared to other Container hosting options in Azure

With all the computing solutions Microsoft Azure is offering, it can be hard to choose the right one. When should you choose Functions, Web Apps, Container Instances, Kubernetes, or Container Apps? In this section we will help you make the choice.

[1] https://landscape.cncf.io/serverless?selected=dapr
[2] https://pages.xpirit.com/magazine10

Azure Container Instances

Azure container Instances are the simplest way to run a container in Azure. This is great for running certain processes that are not a web application or background worker because Azure Web Apps and Azure Functions offer more functionality in those scenarios. For other applications that are just a single container, Azure Container Instances is a great fit. Another downside of Azure container Instances is they do not have the ability to scale down to 0 instances, so you always have a certain cost.

Azure Function Apps

Function apps are serverless applications that run based on triggers such as http requests, timers or messages in a queue. Azure Functions need the least amount of configuration of infrastructure so you can focus on business logic. For smaller applications or processing jobs this is the perfect solution. The major downside of functions can be its "cold starts", where processing a first request after being idle can take a while. Because of that, we would not recommend it for APIs or hosting user facing web applications.

Azure Web Apps

Azure Web apps are the go-to solution for basic web applications or API's. As a PaaS solution, configuration is quite simple. There is no built-in support for multi region, so that must be managed by you outside of Azure Web Apps. Azure Web Apps also supports running containers and can be a great combination for front ends combined with workers as Function apps or container instances.

Azure Container Apps

ACA can be seen as a "Kubernetes To Go" solution, in which the developer can use a large amount of the power of Kubernetes without the hassle of maintaining a cluster. However, not all Kubernetes functionality is available for the user. An ideal scenario would be the containerization of microservices or any background process with fluctuating load peaks by using KEDA auto scalers. Having a full application in Azure Container apps in the future could combine the best of both worlds comparing it to the features that Kubernetes brings versus the simplicity of the combination of Azure Web Apps & Azure Functions.

Azure Kubernetes Service

AKS is the Kubernetes PAAS offering by Microsoft, in which Microsoft maintains the underlying cluster technologies and virtual machines. This does not mean it does not need maintenance. User management, ingress & egress routing, networking, security, and resource allocations are all things that have to be taken into account by you when using AKS. The learning curve can be steep to start working with Kubernetes, but it does offer a stack that can run almost any application in any technology stack. Combining that, along with scaling and self-healing / auto recovery services Kubernetes provides, makes this is a great option for organizations that want to host business critical applications.

Are Azure Container Apps the future of microservices on Azure?

Azure container Apps (ACA) is currently in public preview. It was announced at Ignite near the end of 2021 and still has some time to go before it will become Generally available (GA).

Since we thrive by the "you build it, you run it" mantra and love building microservice architectures, we're often in a love/hate relationship with Kubernetes. It has so many good features, but they come at a price of added complexity that development teams often can not grasp. Therefore, we love the concept of Azure Container Apps which brings a lot of these features without the complexity. However, Azure Container apps in its current state does have some flaws. We hope these would get solved soon and some of them are already on Microsoft's roadmap.

Some major improvements we would like to see?

Managed Identities

Managed Identities are the way to connect running services to other Azure resources such as databases or queues. At this moment Managed Identities are not supported yet but Microsoft announced this will be available before GA.

Investigating running containers

At this moment the only way to inspect running containers in ACA is through the logging in Log analytics. Microsoft already announced that they are working on a way to improve this. We have the hope this will make investigating issues during development will be a lot easier if that is possible.

Advanced traffic shaping

The current revisions within ACA will allow you to shape traffic to each revision. The only way to do this is by setting a certain % of the traffic towards it. We think this is a nice idea but in practice almost nobody uses it this way. It would be a lot better if we could shape the traffic in more advanced ways like sending traffic with certain http request headers to 1 revision or the other or other options all defined in the SMI-Spec[3].

Regional failovers

Azure Container apps currently have no options for regional failovers when there is an outage. One of the benefits of Kubernetes is you can have a cluster expand over multiple regions and it can handle failure of the compute in a zone or region. You could deploy the same ACA in 2 regions and put an Azure Frontdoor in front of them to direct the traffic, but it would be nice to have this built into the service especially in countries that focus on 1 region. ACA does automatically deploy to multiple availability zones for high availability so that's a good start.

Limited hardware configuration options

When creating Container Apps, you can allocate CPU and memory resources to them. Currently there are only options ranging from 0.25 CPU cores and 0.5Gi memory to 2 CPU and 4Gi memory. We think this should be more flexible for apps that are not heavy on the CPU but do need more memory or the other way around.

The Future of hosting microservices in Azure?

Azure Container Apps is still in preview. There are a lot of improvements already underway. Time will tell. We are enthusiastic about the movement to a more serverless way of running a Kubernetes-like environment for our microservices. Azure Container Apps is a big step into the right direction. As of this writing, there are just a few features missing that would prevent us from using this in production, such as the lack managed identity. We do believe, if these would be added, this could become a dominant platform for hosting containerized workloads, especially for microservice based applications. </>

Geert van der Cruijsen
Trainer, Digital Kickstarter, Enabler for companies to embrace DevOps, Cloud & improve their engineering culture

xpirit.com/geert

Bas van de Sande
Azure Coding Architect, Consultant, Integrator

xpirit.com/bas

[3] https://github.com/servicemeshinterface/smi-spec/blob/main/apis/traffic-split/v1alpha4/traffic-split.md