Google Cloud Platform

Running Your Modern .NET Application on Kubernetes

by Andy Wu, Solutions Architect, Magenic

Table of Contents

Executive summary

What is a cloud-native application? For the past two white papers in this series, we have covered this subject by applying the principles of a microservice architecture as well as other application modernization techniques, but have we fully achieved the goals we established at the outset of our journey?

Before we answer that, let's first agree on a the formal definition of a cloud-native application: According to the Cloud Native Computing Foundation (CNCF), A cloud-native application should have the following characteristics:

1. **Containerized**. Each part (applications, processes, etc.) is packaged in its own container. This facilitates reproducibility, transparency, and resource isolation.
2. **Dynamically orchestrated**. Containers are actively scheduled and managed to optimize resource utilization.
3. **Microservices-oriented**. Applications are segmented into microservices. This significantly increases the overall agility and maintainability of applications.

With this definition, it would seem we are missing a couple of elements in order to call our modified PetShop a true cloud-native application--containers and orchestration.

In this final white paper of the series, we will introduce these missing pieces and show readers why and how they can be applied to our application so that it can achieve the status of being cloud-native.

Why is it important to build cloud-native applications?

A recurring theme we hear is the call for IT to innovate at the speed of business. As businesses are being impacted by disruptive forces that are occurring at an accelerated rate, correspondingly, IT is being pressured to iterate and release software at a faster pace in order to keep up. When done right, cloud-native architecture represents the best approach we have today to provide the agility, reliability, and scale needed to meet this challenge. For companies that have adopted the cloud as the new default, the cloud-native approach should be a natural choice for the development of green-field applications. Even for brown-field applications, cloud-native architecture should be considered if the migration cost can be justified, because while we cannot predict the future, we can be certain the velocity for change is not going to slow down anytime soon. Hence, if a system is of critical importance to the business, then it should be justifiable to make the necessary investments so that it can respond quickly.

This cloud-native approach is being promoted not because it's fashionable; there are pragmatic motivations to be achieved. The approach works well with continuous delivery to achieve faster time-to-value, it scales well, and it is efficient to operate. Most importantly, when combined with the right application architecture, such as the microservices architecture we have prescribed for PetShop, it can greatly reduce risk in a new way – by going fast but safe.

Many organizations are executing the idea of "You build it, you run it" by leveraging the cloud-native approach. The basic idea behind this phrase is developers who develop code will be responsible for deploying the code right to production as well. That's right, production. And the only reason why organizations can safely execute this idea is because:

a. The scope of microservices is granular so impact will be limited and isolated;
b. With container technology, they can easily roll back if something does go wrong with any given deployment.

This is precisely how the cloud-native architecture achieves the notion of fast-but-safe.

What's the big deal about containers?

Speed and agility are what every company craves in the Internet age. As a software developer, or architect, nothing ever seems to be fast enough for our users. Moreover, speed is addictive. Once you get a taste of it, whether it's from the development, deployment, or operation, you want more of it. In recent years, more and more developers are preaching the phrase "develop and test at the speed of Docker" as THE way to achieve speed with safety. The reason for this is that containers are indeed helping companies across industries accelerate software development and deployment at scale, while reducing costs and saving IT departments time.

Here's how: Prior to the advent of containers, one of the biggest resource drains in the software development lifecycle was the discrepancy between the various application hosting environments along the deployment pipeline. Anywhere along the pipeline from a developer's laptop, to test, staging, and finally, production, there could be any number of configuration differences that could cause the application to run improperly. It could take hours, or even days, to understand and track down the root cause for these discrepancies, and that's simply a terrible waste of time and resource. Containers are revolutionizing this process by providing great environmental fidelity. With the portable nature of containers, you are, by definition, running your system under an identical environment, from development all the way to production! The high adoption rate for containers as a critical part of software development has proven containers should be a fundamental part of any enterprise application development and hosting strategy. Containers not only facilitate an application modernization roadmap, but they prepare one's IT environment for the future.

No doubt about it, we're at an inflection point with containers, similar to where virtual machines were roughly 10 years ago. There is a great deal of momentum behind the technology as

developers, architects, and operations personnel are discovering the benefits it provides. Specifically, containers provide convenient development and packaging tools for developers that spread into IT and operations, dramatically lessening the discrepancies, hence increasing the velocity and accuracy of software releases. That's something we, as a industry, have never had before.

Challenges with running containers at scale

Now that we've established the validity and benefits of container usage, let's take it a step further by discussing how they are typically used in a production environment.

With the advent of microservices, it's not unusual to have tens, hundreds, or even thousands of container instances running across multiple hosts in order to support a particular production workload. It's one thing to spawn a few container instances by hand when we're developing software, but to use containers at scale, in a production environment, we must consider how the following list of requirements can be accomplished:

- Scheduling of container instances: which instance of the VM (or bare metal) should host the new container instance in order to maximize and optimize system resources?
- Monitoring of containers, when a container fails (and they will), what's going to stop the errant instance and restart another healthy instance?
- When load increases, what is going to do the load-balancing by automatically starting more instances of container to take on the extra traffic?
- What if your company needs zero-downtime deployment?

Still think you can manage this by hand? I sure hope not, and this is why container orchestration framework such as Kubernetes, Mesosphere DC/OS, and Docker Swarm exist: to provide some, or all, of this functionality so we can focus on running our applications, safely, and reliably.

The power of ephemeral computing

Before we go further about the tooling needed for managing containers at scale, we need to touch on the notion of reliability for cloud-native applications. Traditionally, we tend to define reliability of a system in terms of its ability not to fail. We thought we could accomplish the Herculean feat of writing the perfect software, designing the perfect infrastructure, and running the perfect operation. I'm not sure about your experience, but I know personally, I have yet to come across any application system that doesn't fail. It's time for us to face the facts: failure is inevitable. Instead of trying to avoid it (if that's even possible), cloud-native architecture actually anticipates it by defining reliability in terms of a system's resilience (i.e., its ability to mask system failures by failing fast and recovering rapidly so any failures are not perceived by the end user). Servers go down, stacks overflow, we have defects in our code—that's just the facts of life. Instead of doing the diminishing return act of hunting down every ever-elusive defect in a system, cloud-native architecture argues our time can be better spent by designing our system to handle failure gracefully and quickly.

The way we do that in the world of containers is to stop treating them as pets, but rather as cattle. Huh you say? The metaphor comes from the idea that if one treats containers as pets (the way we have gotten used to with physical machines and VMs), like naming them, then each instance will likely have special characteristics, making them difficult to replace. If they get sick (malfunction), one spends time nursing them back to health, which will take valuable time and resources. Whereas if one treats containers like cattle, indistinguishable from one to the next, you simply replace the ill with a brand new healthy container instance of the same type. The idea that a container instance can come and go (i.e., they are ephemeral), is how cloud-native applications achieve their reliability and scalability under the covers. Containers expect your code to start up and shut down quickly so the container orchestrator can do its job: scale properly, mask failures, and maximizes system resource. Make no mistake about it, this is a fundamental shift in thinking, where architecting for failure is the rule rather than exception.

What is Kubernetes?

Even though there are a number of container orchestration frameworks available in the marketplace, in recent years, Kubernetes has really sprung ahead of the others.

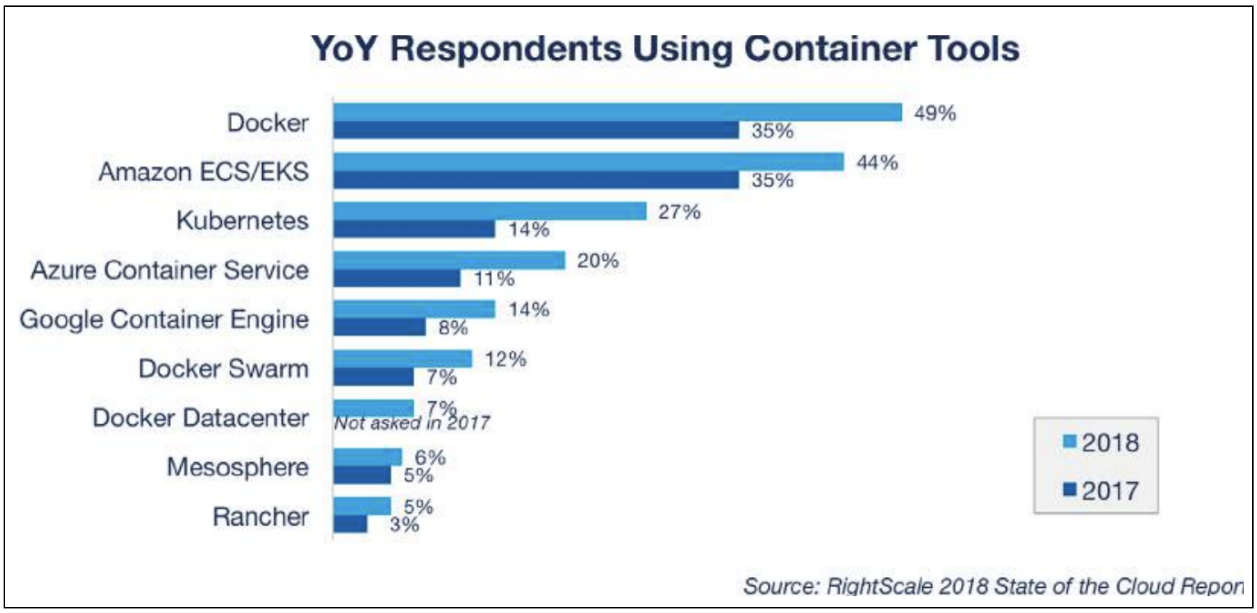
Kubernetes is an open source platform for automating deployments, scaling, and operations of application containers across clusters of hosts, providing a “platform” for container-centric infrastructure. It addresses all the challenges that were described above. Among its key features are:

- Automated deployment and replication of containers
- Online scale-in and scale-out of container clusters
- Load balancing over groups of containers
- Rolling upgrades of application containers
- Resiliency, with automated rescheduling of failed containers (i.e., self-healing of container instance)
- Controlled exposure of network ports to systems outside of the cluster

This open-source software project was started in 2014 by Google and builds upon Google’s internal cluster management system, called Borg, that has been refined internally at Google for more than a decade. To give you an idea of Kubernetes’ popularity and the scale of its community, as of 2017, Google cited that the Kubernetes project has received more than 400 man-years worth of contribution since its inception!

While it’s not a simple system to learn, its learning curve has been greatly reduced by the wealth of community contributions such as in-person meetups in every major city in the world, KubeCon (a developer and user conference for Kubernetes), tutorials, blog posts, support from Google, and finally, an official Slack Channel. Most of all, it’s an officially supported technology from every major cloud provider--Google Cloud, Microsoft Azure, and AW--making it the leading and most ubiquitous container orchestration framework available in the market.

We will finish up the argument for Kubernetes by showing its rapid adoption rate with a recent survey done by RightScale:

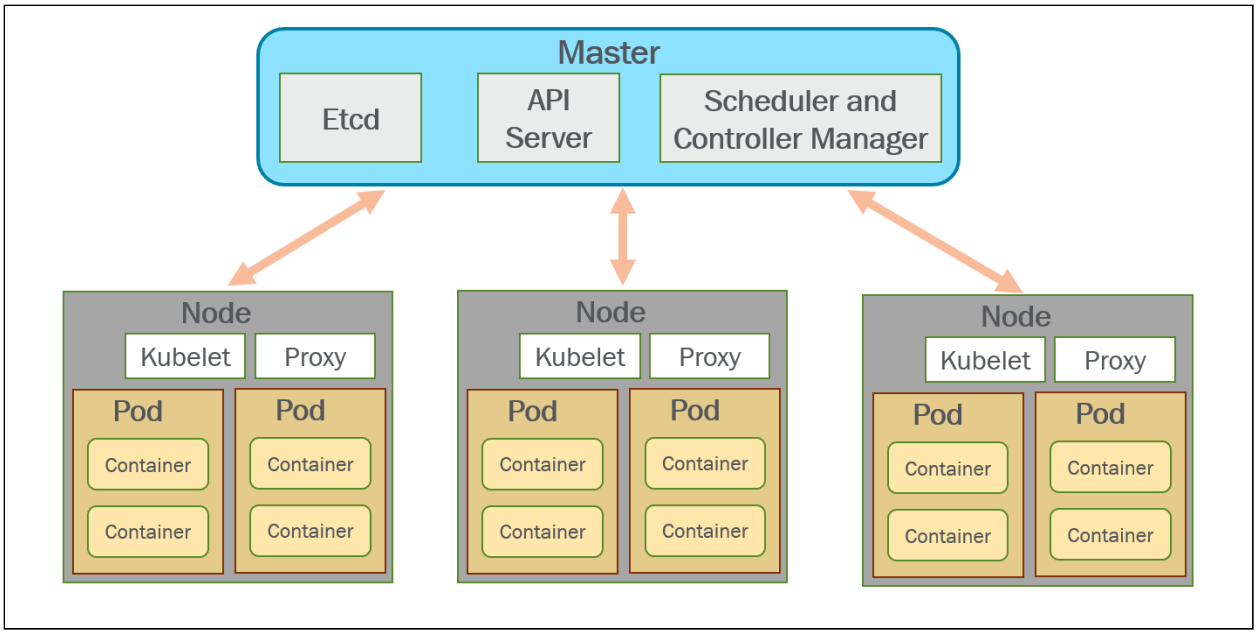


(Source: RightScale 2018 State of the Cloud Report™ © 2018 RightScale, Inc. All rights reserved. This work by RightScale is licensed under a Creative Commons Attribution 4.0 International License.)

Container-as-a-Service (CaaS), the new kid on the XaaS block

As mentioned earlier, Kubernetes is open and modular, you can run it on a VM or bare metal, on-premises or in the cloud. A deep dive on the technical details of this system is well beyond the scope this paper, but the amount of documentation available is more than plentiful, and I would suggest one starts by looking at the official documentation.

To give one an idea of what is involved in setting up a Kubernetes cluster, here is a high-level pictorial view of what the system looks like:



Let's just say it's not as simple as just setting up a few VMs and installing some software on top of it. As a side note, if one is really interested in learning what's involved in setting up a Kubernetes cluster by hand, from scratch, there is an excellent GitHub project put forth by subject matter expert—Kelsey Hightower, a Google Developer Advocate—that goes through the setup in all its gory detail.

Given its complexity, the question is should one be setting up Kubernetes by hand? Moreover, should one devote ongoing resource to make sure it is kept up to date and well tuned? For most organizations, the answer is clearly no.

As containers grow in ubiquity, all major cloud providers are offering Container-as-a-Service (CaaS) as part of their XaaS services. CaaS is a form of container-based virtualization in which container engines (such as Docker), orchestration, and the underlying computing resources are delivered to users as a service, over the Internet, completely self-provisionable from a cloud provider. CaaS helps organizations run containers at scale, alleviating developers and system administrators from worrying about the container platform their applications depend on, enabling them to focus on development of their container-based applications.

Kubernetes on Google Cloud Platform

As a CaaS, Google Kubernetes Engine (GKE) provides a managed environment for deploying, managing, and scaling your containerized applications using Google's infrastructure.
This is a great value-add to Kubernetes for organizations, providing advanced cluster management functionalities, such as:

- Google Cloud Platform's load-balancing for Compute Engine instances
- Node Pools to designate subsets of nodes within a cluster for additional flexibility
- Automatic scaling of your cluster's node instance count
- Automatic upgrades for your cluster's node software
- Node auto-repair to maintain node health and availability
- Deep integration with Stackdriver for logging and monitoring

For those companies that have aspirations to run modern, cloud-native applications and operations, choosing GKE to host their container workload is a natural fit.

How to containerize PetShop

There are no shortage of detailed tutorials, labs and other resources available on the web on how to create Docker containers for your .NET Core applications, so for the sake of brevity, I will simply highlight what to take note of when one is intending to deploy their Docker containers to run on Google Cloud Platform. If one is interested in going through a step-by-step tutorial on how to containerize an asp.net core application and have it be hosted on Google Kubernetes Engine (GKE), I'd highly recommend this codelab: "Deploy ASP.NET Core app to Kubernetes on Kubernetes Engine".

Like all containerization processes, we start with a Dockerfile for each of the three microservices we have for our application: Product, Profile, and Order.

Dockerfile for the Product Service:

```
FROM gcr.io/google-appengine/aspnetcore:2.0
COPY . /app
ENV ASPNETCORE_URLS=http://*:${PORT}
WORKDIR /app
ENTRYPOINT ["dotnet", "ProductServiceCore.dll"]
```

Dockerfile for the Profile Service:

```
FROM gcr.io/google-appengine/aspnetcore:2.0
COPY . /app
ENV ASPNETCORE_URLS=http://*:${PORT}
WORKDIR /app
ENTRYPOINT ["dotnet", "ProfileServiceCore.dll"]
```

Dockerfile for the Order Service:

```
FROM gcr.io/google-appengine/aspnetcore:2.0
COPY . /app
ENV ASPNETCORE_URLS=http://*:${PORT}
WORKDIR /app
ENTRYPOINT ["dotnet", "OrderServiceCore.dll"]
```

Note the usage of `gcr.io/google-appengine/aspnetcore:2.0` as the container's base image; this is the official Docker image that has been optimized for running ASP.NET Core apps in Google App Engine as well as Kubernetes Engine. It's highly recommended that this image be used when containers are to be deployed to GAE and GKE.

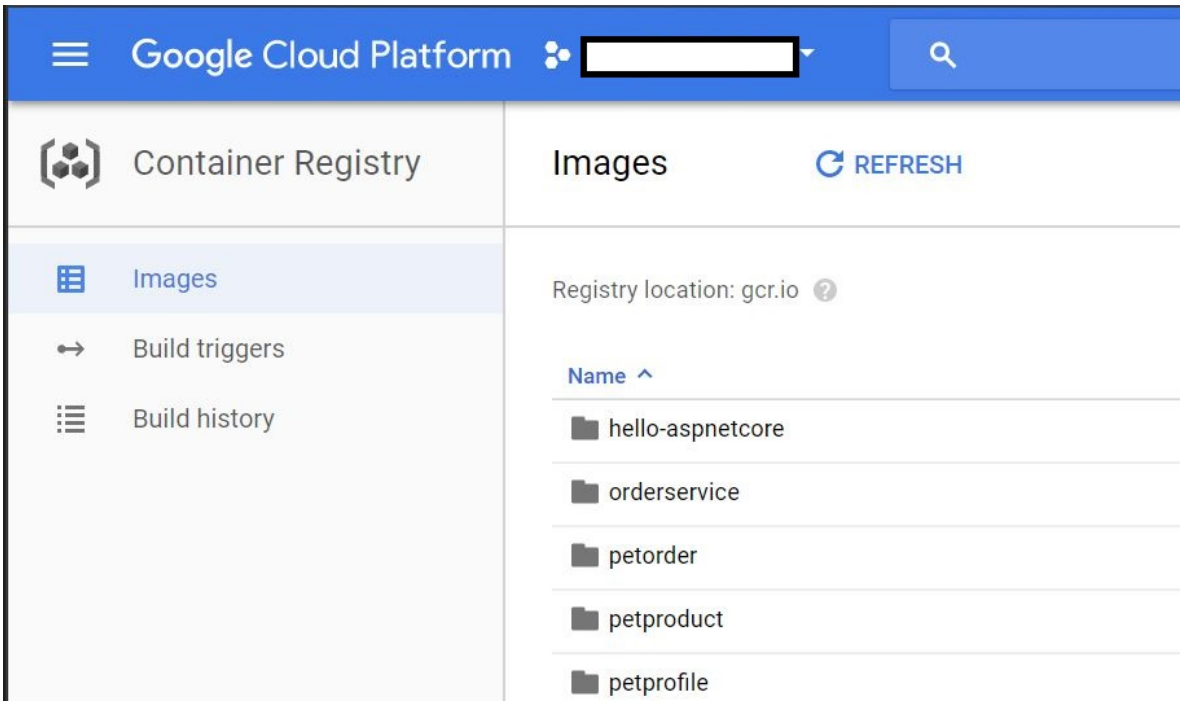
Once the Dockerfiles are created, one can build the images by running the docker command:

```
Docker build -t gcr.io/YOUR_PROJECT_ID/orderservice:v1
```

To push the built images onto the Google Container Registry, a private repository for your Docker images accessible from every Google Cloud project (but also from outside Google Cloud Platform), one uses the following command :

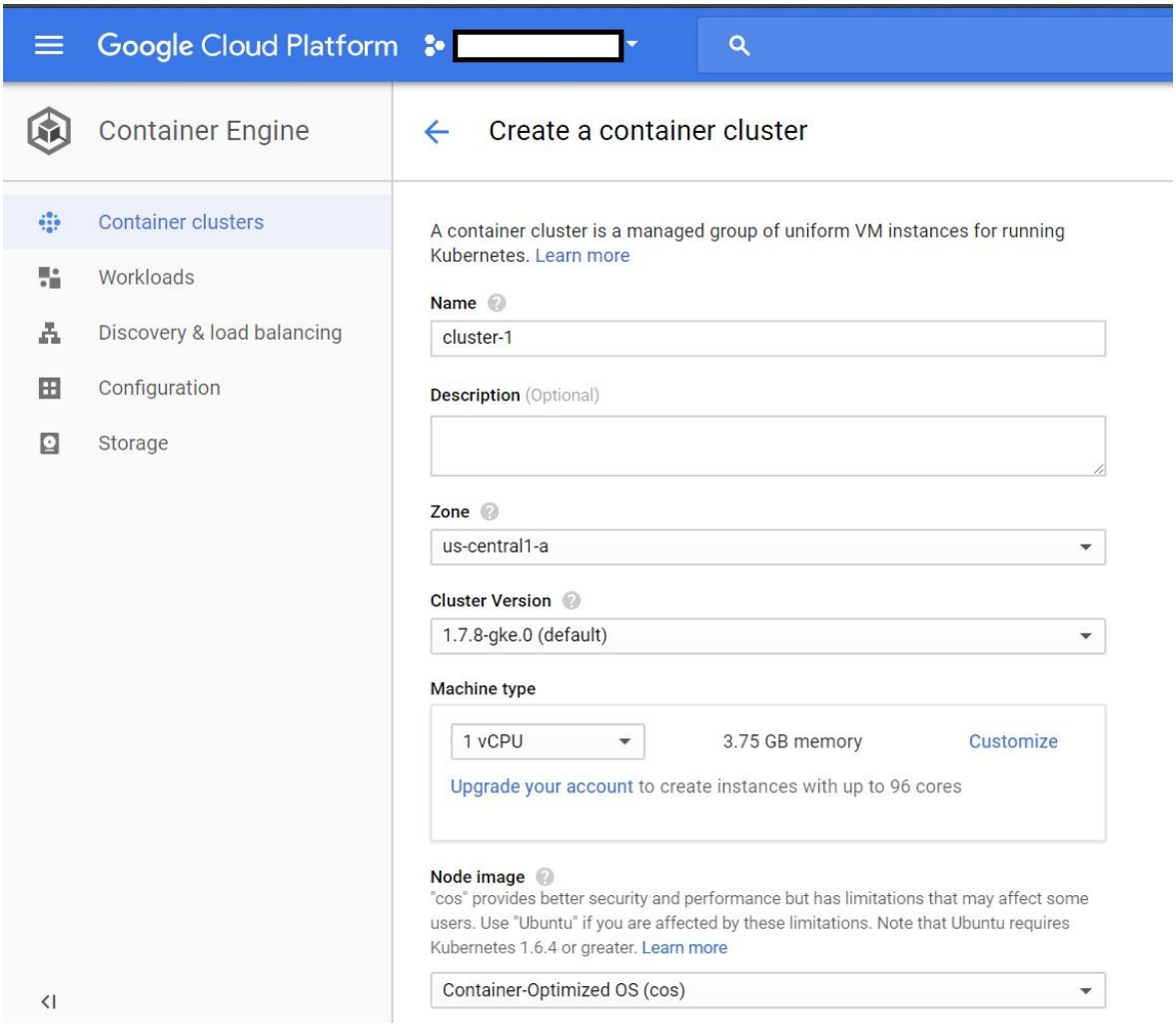
```
gcloud docker -- push gcr.io/YOUR_PROJECT_ID/orderservice:v1
```

Once the images are pushed onto the registry, one can browse them via the web console:



Kubernetes setup for PetShop

Now that we have our Docker images created and uploaded onto the Google Container Registry, we are ready to create the Kubernetes cluster to run them. One can choose to create the cluster by either navigating to the web console, and fill in the cluster creation form:



Or, simply create the cluster via the gcloud command line interface:

```
gcloud container clusters create petshop-dotnet-cluster \
               --num-nodes 3 \
               --machine-type n1-standard-1 \
               --zone us-east1-c
```

With the cluster created, we are now ready to deploy our services, and for that, we will use the `kubectl` command line utility (setup by the gcloud SDK).

First we need to create pods. A Kubernetes pod is one or more containers that are tied together for the purpose of cohesion and administration. We can create pods using the kubectl run option (note the port of 8080, this is the default port used by the Google App Engine based image):

```
kubectl run profileservice \
--image=gcr.io/YOUR_PROJECT_ID/profileservice:v1 --port=8080
```

By using the run option, we have created a deployment object, the recommended option for creating and scaling pods.

To view the deployed deployments:

```
kubectl get deployments
```

```
PS C:\k8s> kubectl get deployments
NAME             DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
orderservice     1         1         1            1           3d
productservice   1         1         1            1           3d
profileservice   1         1         1            1           3d
```

To view the pods that are created under these deployments, use the following command:

```
kubectl get pods
```

```
PS C:\k8s> kubectl get pods
NAME                              READY     STATUS    RESTARTS   AGE
orderservice-1303125414-t4qxp     1/1       Running   0          3d
productservice-1613570133-kgkbp   1/1       Running   0          3d
profileservice-768811045-fzksq    1/1       Running   0          3d
```

Now that we have pods created and running, we need to expose them to the outside world since by default, pods are only accessible by internal IP addresses within the cluster. To do that, we need to create Kubernetes service, and specify LoadBalancer as its type. This requests the underlying infrastructure to create a load-balanced front end with a publicly addressable IP address:

```
kubectl expose deployment productservice --type="LoadBalancer" --port=80 --target-port=8080
```

To get the IP address of the load-balanced service:

```
kubectl get services
```

```
PS C:\k8s> kubectl get services
NAME             CLUSTER-IP      EXTERNAL-IP       PORT(S)        AGE
kubernetes       10.31.240.1     <none>            443/TCP        59d
orderservice     10.31.255.206   35.192.84.186     80:30232/TCP   3d
productservice   10.31.254.100   146.148.100.155   80:31027/TCP   3d
profileservice   10.31.240.251   35.184.95.183     80:30349/TCP   3d
```

With these IP addresses in hand, we are ready to go back to the PetShop Web Site's web.config and input these addresses so the web site can properly connect to these microservices:

```
<appSettings>
  <!-- Web Services -->
  <add key="ProductBaseURL" value="http://146.148.100.155/"/>
  <add key="ProfileBaseURL" value="http://35.184.95.183/"/>
  <add key="OrderBaseURL" value="http://35.192.84.186/"/>
   <add key="Event Log Source" value=".NET Pet Shop 4.0"/>
</appSettings>
```

Demo of Kubernetes autoscaling capabilities

We now have a fully cloud-native version of PetShop complete with microservices, modernizations, containers, and that's hosted in Google Kubernetes Engine (GKE). Let's show off those capabilities by doing a demo of this powerful new implementation.

What we will do in this demo is to show how easy it is to enable autoscaling at the pod level as well as the Kubernetes node level.

Quick primer:

- A pod represents a unit of deployment: a single instance of an application in Kubernetes, which might consist of either a single container or a small number of containers that are tightly coupled and share resources;
- Nodes in Kubernetes are the VM or server that host pods.

Kubernetes provides built-in support for pod-based autoscaling called Horizontal Pod Autoscaling (HPA). HPA works by automatically scaling the number of pods in an application based on observed CPU utilization. Since pods are hosted on Kubernetes node(s), there are only a finite number of pods that can be spun up before a node runs out of resources for creating new pods if load continues to grow. To fully realize the goal of container-based autoscaling, one also needs a node-based autoscaling mechanism to work in conjunction with HPA to scale the number of nodes. GKE offers an unique feature called Cluster Autoscaler that works by automatically resizing the Kubernetes clusters based on the running workload demands. Once Cluster Autoscaler is enabled, GKE will automatically add a new node to a cluster once a node runs out of resources to fulfill a new pod creation request. Better yet, it continues to monitor usage and if the load diminishes, GKE will correspondingly rightsize the number of nodes to match the load, optimizing resources and cost. Conceptually, this is the classic cloud elasticity model at play, with the added container layer.

For the convenience of the readers, we have created a video to demonstrate this set of technologies, and you can access it here: https://www.youtube.com/watch?v=w0FMVP37JE0

Conclusion

We made it! We have completed the journey for taking the monolithic PetShop application and re-architecting and modernizing it to be a cloud-native microservices-based application. Through this journey, we took a legacy system that was hard to maintain and modernized it into an efficient, cloud-native system that is ready to take full advantage of what cloud computing has to offer. Perhaps best of all, we made it much easier to maintain going forward.

The scenario we chronicled is not unique. Many companies are saddled with systems that are not easily sunset or wholesale replaced, yet in many cases, they are critical systems for the company (i.e. the applications companies rely on for profits). In the age of digital transformation, systems must adapt faster than ever before to meet demanding capability and performance needs. A cloud-native approach of system delivery empowers organizations to meet these challenges while accelerating the pace of innovation, all in a risk-averse and sustainable way.