<>Lessons Learned in the Cloud</>

This white paper takes the reader on a journey from hosting an application on their infrastructure to the advantages of running that application in the Cloud.
Cloud computing allows for the outsourcing of many operational tasks, freeing users to focus on the business logic. Platform as a Service (PaaS) solutions often come with vendor lock-in and limited customization, so it's usually better to rely on open-source standards such as Kubernetes. This paper introduces readers to the Managed Kubernetes solution: OVHcloud - a sweet spot between vanilla Kubernetes and PaaS.

Index

In the beginning there was no cloud.

Ever wondered what running an application in the Cloud and making a pizza have in common? Imagine someone living in historic Italy and dreaming up the perfect recipe for a pizza before it was ever invented: the flavour of all ingredients and exactly how the pizza is to be made is all there in the dream - all that's needed is a stone oven, flour, cheese, tomatoes, spices, and olive oil (and perhaps a few more ingredients if the dream featured something more complicated than a Margherita!). Roll the dough, cover it with toppings, and put everything into the oven, et voilá - eight minutes later, there's the pizza. Hosting an application in the Cloud using PaaS is essentially the same process, though the stone oven has already been built, and the ingredients sourced for you.

That dreamer in early Italy wouldn't have all these things of course. He would need a cow to produce milk to transform into cheese, a process that takes time and effort. The same applies to the flour, oil, spices, and even the tomatoes, and of course he would have to construct his stone oven to be able to bake the pizza of his dreams. Doing all this from scratch means he would not benefit from economies of scale. Our dreamer wouldn't be able to make use of the expertise others have acquired over the centuries in areas such as oven manufacturing, livestock farming and agriculture. He would need to take care of his cow if it got sick as well as repair his oven if it broke, using tools he would need to create for himself.

This latter scenario is like an on-premises data center prior to the advent of the Cloud as we know it today. Without the Cloud, our situation is similar to that of the dreamer in our story: we have a group of computers or physical hosts with real sockets CPU, RAM, and storage; lots of expertise is needed to implement a secure application that can be used by a considerable amount of people, and; to run the application requires application runtime preinstalled on the OS (which also needs to be installed and managed).

Virtualisation is an important concept that drives the Cloud. Virtualisation introduces the concept of software-based representation of a computing resource or a host. Without virtualisation to launch an additional application instance, a new host would need to be added, with the application running on, connecting it to the existing network (which means dealing with networking, hardware, and even the cooling of the new hardware resource).
Virtualisation is made possible by the hypervisor installed on the physical host; instead of ramping up a new physical host, this enables us to add another virtual host or virtual machine (VM), without dealing with hardware. If a physical host needs to be replaced by another more powerful machine, the VM machine can just as easily be transferred to another physical host. In other words, virtual machines are the essential abstraction layer that allows developers and operators to bypass dealing with the hardware.

Virtualisation didn't stop with virtual machines; state-of-the-art systems use Linux containers, which are themselves lightweight virtual machines. The key difference is that a VM virtualises an entire machine down to the hardware, while a container only virtualises software layers above the operating system level.
Where precisely does the Cloud begin? The meme "The Cloud is what we used to call 'someone else's computer'" has a lot of truth in it, but of course, the real story is much more complicated than that. Using the Cloud means having access to cumulative expertise and best practice in networking, storage, and computing, as well as benefiting heavily from economies of scale, thus reducing costs.

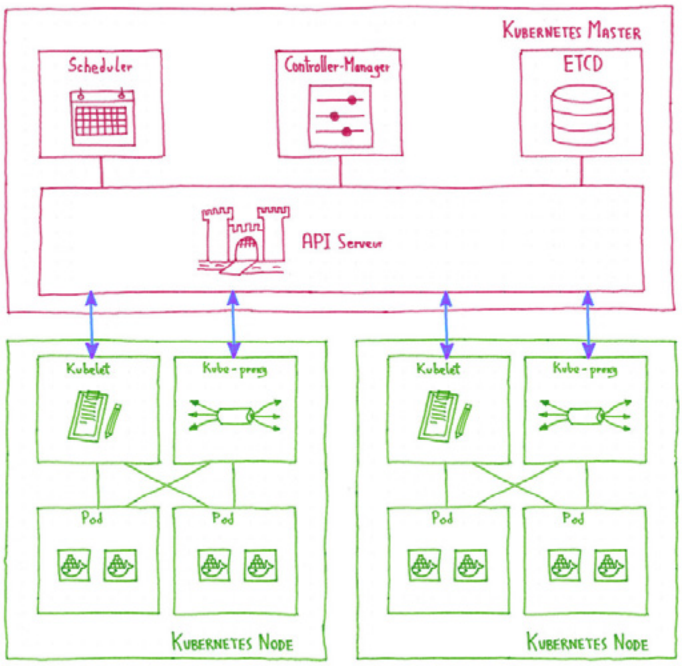
Kubernetes to the rescue

From Traditional to Virtual to Container Deployment

Moving towards container technology simplified simple replication and deployment of containerized applications, but it also made independence from the actual OS an application is running on much simpler to achieve. If the container runs the application and incorporates all necessary configuration parameters, all that is needed is to ensure the correct container is shipped to the relevant infrastructure.
But how can containers be distributed among virtual machines with a guarantee that the required network policy and resource utilization requirements are applied? This requires the implementation of an orchestration engine such as Kubernetes (also referred to as K8S in this white paper), which is now the de-facto standard to run containers.

Kubernetes basics

Figure 1 K8S cluster and its components



Before continuing with the “why” and “how”, here is an overview of the architecture of a Kubernetes cluster. The cluster typically consists of one or more manager (master) and worker nodes (a node is more or less a simple virtual machine running some flavour of Linux). A master node is also referred to as the control plane and runs all software components of a worker node as well as some dedicated control plane components (making it possible to run pods on a single master, although that means renouncing the concept of a cluster). The dedicated components of worker and master nodes are:

Master components

+ API Server: the main entry-point for controlling the cluster, typically accessed via the kubectl command line tool (custom clients can also be implemented to access the API)
+ Scheduler: receives the API request to run a new pod and finds a suitable worker node for that purpose
+ Controller-Manager: a component that runs controllers, controls loops, watches the state of the cluster and moves it towards the desired state
+ ETCD: datastore which retains the current state of the cluster

Worker components

+ Kubelet: makes sure the containers described in PodSpec are running and healthy (interacts with the container runtime)
+ Kube-proxy: creates and manages local firewall rules & networking configurations to expose containers on the network

Why choosing Kubernetes

Returning to our pizza example will help to explain why a Managed Kubernetes as offered by OVHcloud is the choice to make. A good Italian restaurant will probably use home-made dough to make pizza and pasta. Why? To control the process and the quality of the food as much as possible. Fair enough, but it opens up the question: in that case, should a restaurant have a buffalo farm to make its own mozzarella as well as a field to grow the grain, as it affects the quality of the flour? Some restaurants might do that, but for most, it's a question of the tradeoff between control and economic considerations. To have that degree of control means having expert personnel with the required skill set. If the team is made up of just one person, and their skills are not those of a chef, it might be worth considering using frozen pizza in the business. In terms of the Cloud, this means entering a higher level in the cloud computing stack, such as PaaS (this level or so-called cloud service models will be discussed in the next section).

So, when is Kubernetes the right choice? If more than one of the statements below is true for a business, it might be the proper choice:

→ The business is willing to operate a couple of nodes running the K8S ecosystem

→ The people to handle the configuration and maintenance of the cluster are in place

→ The business offers more than one service (maybe not just pizza!)

→ Automation of a homogenous service deployment is needed

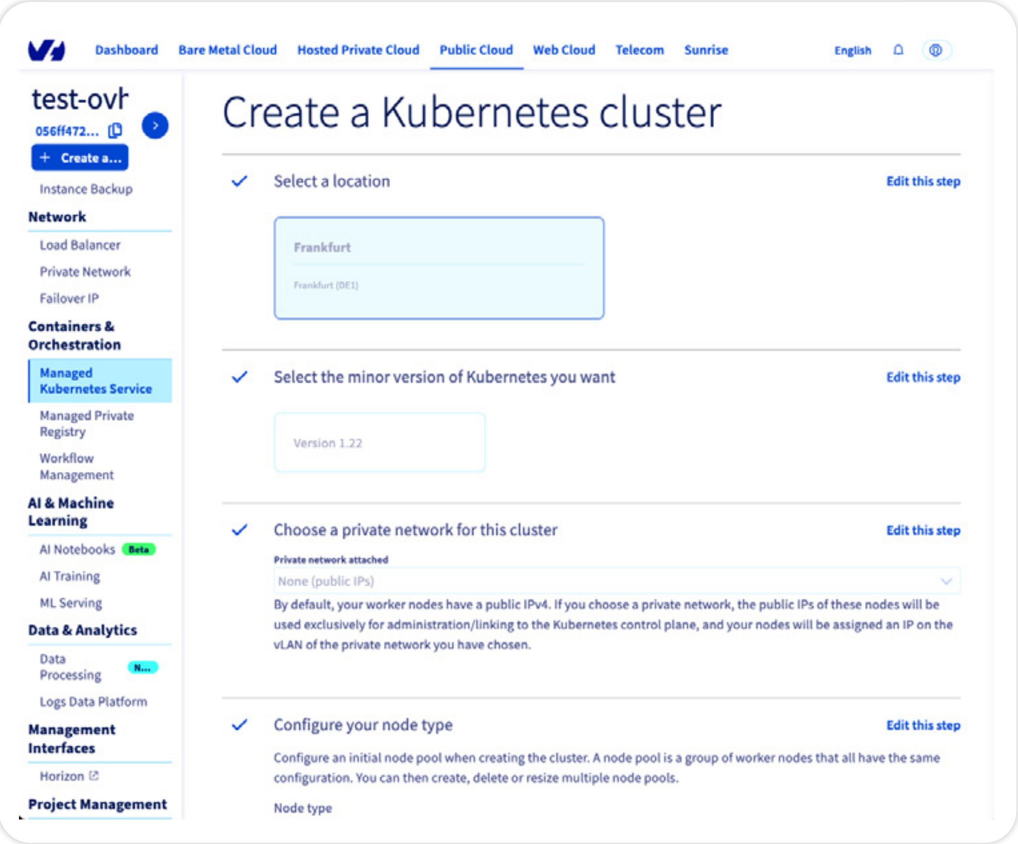
→ The business needs to be cloud-provider-agnostic

How to use Kubernetes correctly

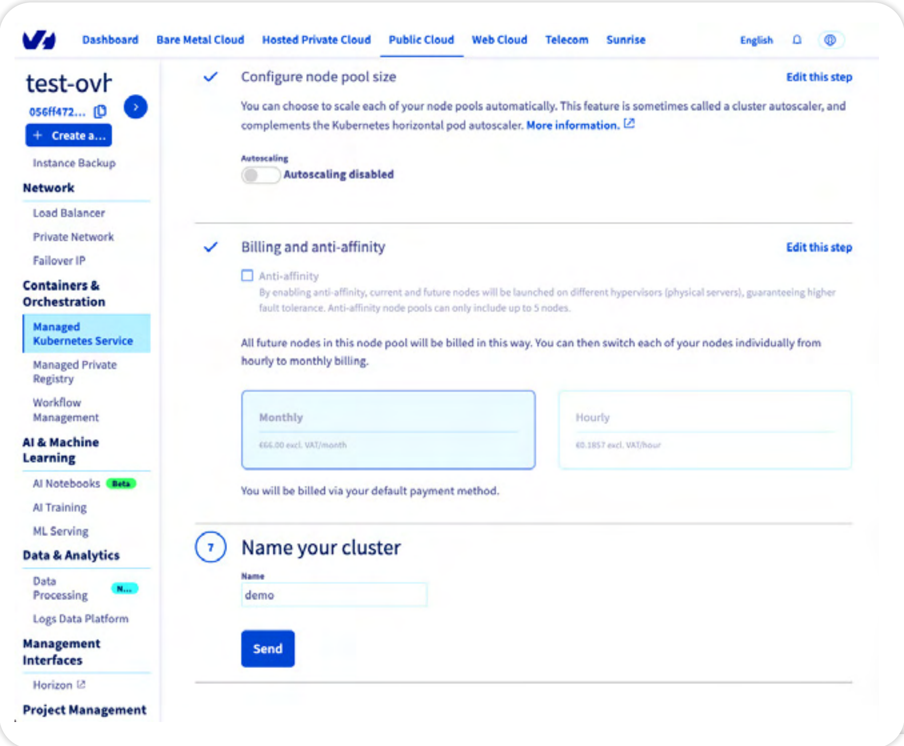
One of the quickest ways to get started with Kubernetes is by using a managed cluster, as offered by OVHcloud. This relies on the best practice built into OVHcloud and saves time and money for business applications - actions such as setting up a cluster or updating it to the latest Kubernetes version require only a few clicks.

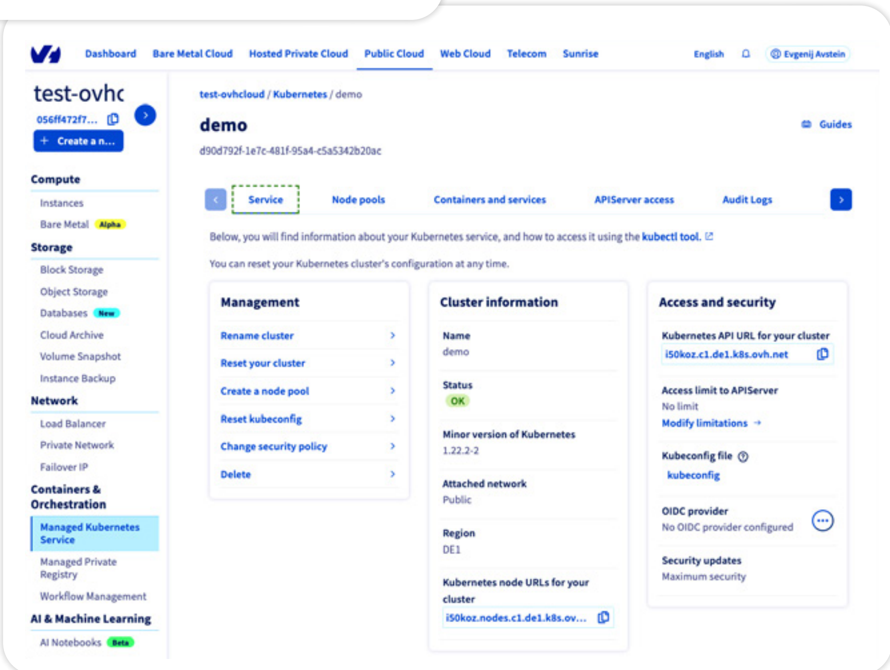
→ *Creating managed cluster*

Spinning up the K8S cluster in OVHcloud only requires a few configuration parameters that the developer can provide in just a few minutes using the OVHcloud Control Panel, as shown below:



The managed K8S solution at OVHCloud is completely free of charge. More precisely, the team at OVHCloud addresses deployment, hosting and updates of all master components as well as the maintenance of the nodes (or virtual machines and the hardware running them). Users need only pay for the resource consumption of the nodes in use.





Creating self-managed cluster

Of course there is still the option to create a cluster oneself if the team has the required skills. This involves the following steps (2):

01 Prepare servers or nodes to run Kubernetes (spinning up the virtual machines; OVHcloud allows users to choose different options, from general-purpose to storage optimized instances), this requires at least two (master and slave) Ubuntu servers.

02 Set up each server to run Kubernetes: install the following software components on each node:

a) Kubelet: this component receives specs for container configuration and invokes the underlying container engine as previously mentioned.

b) Kubeadm: command-line tool with the following commands:

0. kubeadm init: configure current node as master (also called control plane)

1. kubeadm join: configure the current node as a worker (slave) and join it to the cluster

c) Kubectl: central command-line tool to interact with the Kubernetes cluster

03 Set up K8S master using kubeadm

04 Join Nodes to K8S Cluster using kubeadm

05 Set up Networking: Install K8S Add-On enabling networking (Add-Ons are pods and services that enable certain features of the cluster).

Note: the actions above are just the steps needed to set up a cluster (maintenance such as updating or security patching requires other steps and skills).

→ *Using terraform*

As described above, a K8S can be set up using the OVHcloud Control Panel from a browser in just a few minutes. If the enterprise in question expands, it might become desirable to reproduce this process for the different environments in the business - development, quality assurance, acceptance testing, pre-production, and last but not least production. Furthermore, each of the teams responsible for a particular domain might eventually have their own K8S cluster running all the services (or microservices) that form the application. In that kind of scenario, programmatic deployment of K8S clusters might be the ideal. The advantages of this are clear: it's a repeatable process that saves time and thus, cost. This is where infrastructure as code (IaC) comes into play - more explicitly, Terraform. Terraform is an open-source infrastructure code created by Hashicrop in 2014 that allows users to build, change and version control the infrastructure. Enabling Terraform to manage resources on OVHcloud requires the generation of credentials that allow Terraform to call the OVHcloud API endpoints. These credentials need to be provided to the Terraform scripts creating the cluster (a tutorial on this can be found here).

→ *Kubernetes resources*

The process of setting up a new K8S cluster has already been discussed, but what are the resources most commonly seen in a K8S cluster:

Namespace. This enables the segregation of resources in a K8S cluster. Typically, several teams have access to the cluster so it is important to ensure that the resources are isolated from each other. Not all resources are namespaced resources (cross-cluster resources cannot be assigned to a certain namespace; the best example of such a resource is the namespace resource itself)

Pod. A Pod is the smallest, most basic deployable unit in K8S and the abstraction (or wrapper) over a container. A pod consists of one or more containers sharing an IP address, access to storage, and namespace. The architecture allows one process per container. Multi-container pods are very common. A secondary container has several roles:

+ Sidecar: enhances the primary container for the instance logging container to collect data as used by monitoring tools such as Prometheus or fluentd
+ Adapter: modifies the output of the primary container to align with an external format
+ Ambassador: used for communication with outside resources, typically using a local proxy connection or a reverse proxy

The containers inside a pod can use IPC, loopback interface (also known as localhost), or the shared file system to communicate with each other.

Deployment. The deployment resource is used to provision multiple instances of the same pod in the cluster. This K8S object makes sure that the desired replica count and the actual number of pods are equal by running a control loop. Under the hood, the deployment operator uses another K8S resource called ReplicaSet. Another advantage of using a deployment is the ability to perform rollbacks to any previous version of the application.

ConfigMap. As the name suggests, this is used to store data as key-value pairs in the K8S cluster and can be accessed from a pod. This offers the great advantage of being able to externalize the configuration of an application while having different configurations dependent on the current environment. If attached to a pod, a ConfigMap can either be used as a source for the environment's variables or be mounted as a file system. In the latter use case, the resulting file names represent the keys and the content of the files in the values of the ConfigMap.

Secret. Provide secret configuration parameters such as a password to a pod. From a usage perspective (if attached to a pod), this is almost equivalent to using a ConfigMap. The main difference arises when viewing the resulting resource (or the YAML representation) in the cluster: the data is never shown plainly but only encoded.
Of course, if a YAML file containing a secret definition is committed to a git repository then the passwords it contains are exposed to potential malicious users. In this case, SealedSecrets are used. SealedSecret definitions are committed to the git repo containing encrypted passwords which are only decrypted when applied to the cluster (so never committed to a git 'plain').

Persistent Volume and Persistent Volume Claim. The most basic type of volume in K8S is emptyDir (not to be confused with persistent volumes) which allows a container to store data at pod level. This can be utilized for inter-container communication but is not suitable for persistent storage, since the volume is attached to the pod lifecycle (if the pod dies, the data is gone). Persistent volumes as such cannot be directly used by pods (as emptyDir volumes can). Persistent volumes are created by cluster admins. PersistentVolumes are cluster scoped and are consequently another type of resource that does not belong to a particular namespace. They represent storage made available to the cluster by means of filesystem or another storage solution. A persistent volume claim requests (or claims) one part or the whole storage from a PersistentVolume. A PersistentVolumeClaim is the actual resource used from the pod to mount a volume. The latter design approach allows a decoupling from actual storage implementation and consequently leads to a loose coupling between cluster administration tasks (such as preparing storage for the cluster) and those tasks closer to the application deployment process and thus also to the developers.

Service. Kubernetes resource used to expose pods to (external) traffic.
The most commonly used types of service are the following:

+ ClusterIP
+ NodePort
+ LoadBalancer

Routing external traffic to your application inside the K8S cluster.
As soon as an application is running inside a pod, the question becomes how to route traffic to it. By default, all pods in a cluster can speak to each other, so other pods can make curl calls to your application using its internal IP.

Example:

Here is an illustration showing how to access a pod from another pod. Firstly, define a simple pod using the nginx image as follows:

```
apiVersion: v1
kind: Service
metadata:
  creationTimestamp: "2022-04-17T14:16:13Z'
  labels:
    run: hello-world
  name: hello-world-service
spec:
  ports:
  - port: 8080
    protocol: TCP
    targetPort: 80
  selector:
    run: hello-world
  sessionAffinity: None
  type: ClusterIP
```

Create this pod in the OVHcloud cluster and get its IP:

```
➜ kubectl create -f hello-world-pod.yaml
pod/hello-world created

kubectl get pod -o wide
NAME         READY   STATUS    RESTARTS   AGE    IP         NODE                         NOMINATED NODE   READINESS GATES
hello-world  1/1     Running   0          3m11s  10.2.0.5   nodepool-xy <none>         <none>
```

Create and run the temporary pod with a curl call fired to the IP of the first pod:

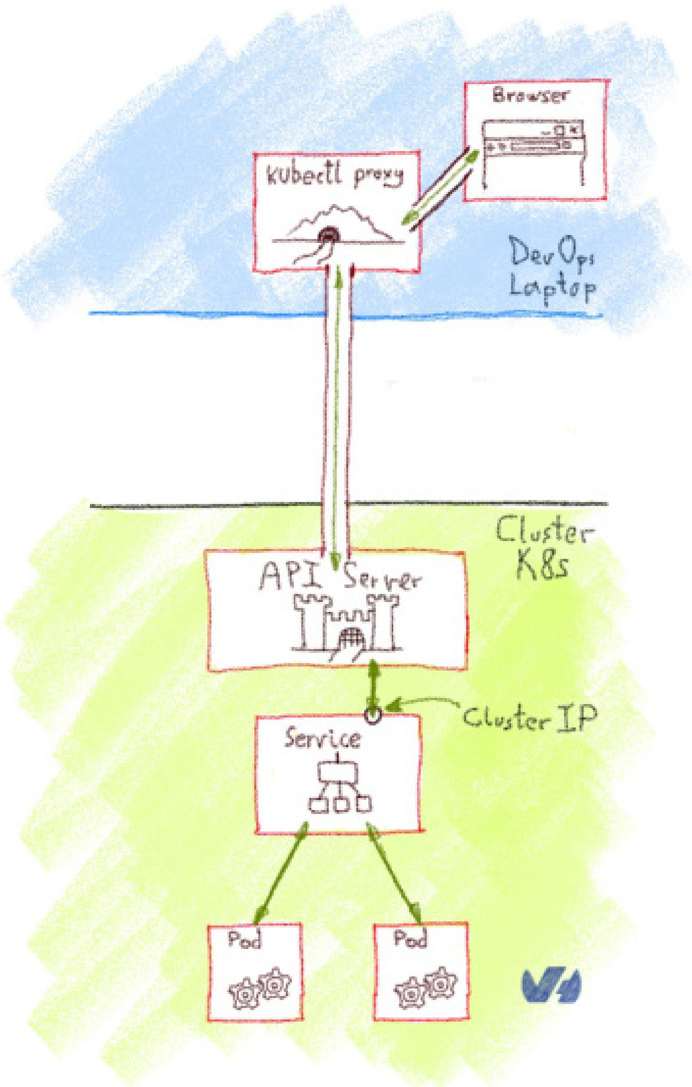
```
kubectl run temp --image=nginx --rm --restart=Never -it -- curl 10.2.0.5
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
html { color-scheme: light dark; }
body { width: 35em; margin: 0 auto;
font-family: Tahoma, Verdana, Arial, sans-serif; }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>

pod "temp" deleted
```

As this makes clear, every pod can 'speak' to any pod in the cluster by default.



ClusterIP: Use of a private IP by the pod is discouraged if the goal is to achieve decoupling from actual IP addresses in use. The reason is obvious: the pod might fail and be relaunched again, assigned to a different IP. That creates a demand for a service that routes the traffic to the pod regardless of the current IP. This can be accomplished using a service resource; service and pod are bound to each other using labels instead of hardcoding the IP. The service of type ClusterIP cannot be accessed from outside the cluster out of the box, but there is the option to create a proxy to the API server (which runs inside the cluster) and 'pretend' that the request is not from outside the cluster, thus allowing access to the application, as shown in the picture below:

Example:
Here is how it can be done, illustrated with a hello world pod.
Firstly, create a service with the following definition:

```
apiVersion: v1
kind: Service
metadata:
  creationTimestamp: "2022-04-17T14:16:13Z"
  labels:
    run: hello-world
  name: hello-world-service
spec:
  ports:
  - port: 8080
    protocol: TCP
    targetPort: 80
  selector:
    run: hello-world
  sessionAffinity: None
  type: ClusterIP
```

Creating the above service enables calling the service inside the cluster using cluster intern DNS from a temporary pod, as follows:

```
kubectl run temp --image=nginx --rm --restart=Never -it -- curl hello-world-service:8080
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>…
```

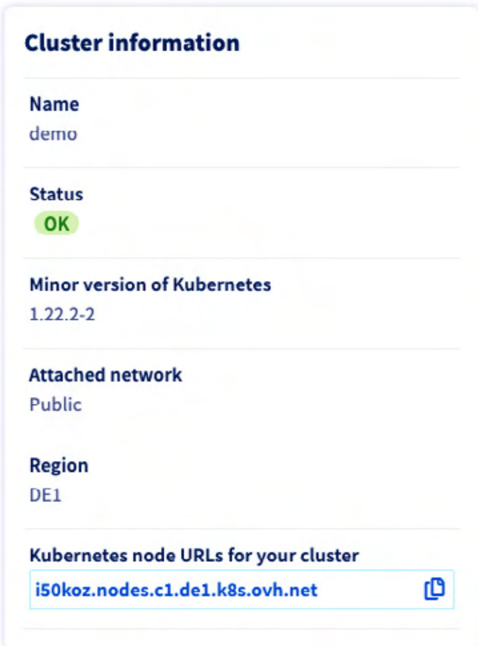
NodePort. If it is necessary to go beyond this and reach the pod from outside the cluster, one of the options is to use a NodePort type of service. The creation of NodePort includes creating a ClusterIP to which the traffic is diverted. The NodePort itself makes sure there is a connection between a high-number port (default 30000-32767) accessible from outside, on each Node of the cluster, to the ClusterIP using iptables or ipvs. More specifically, the application can be accessed on <NodeIp>:<NodePort>.

Example:
To showcase the NodePort setup all that needs to happen is a change in the type of the service. Switch to NodePort and check which port the service was assigned as follows:

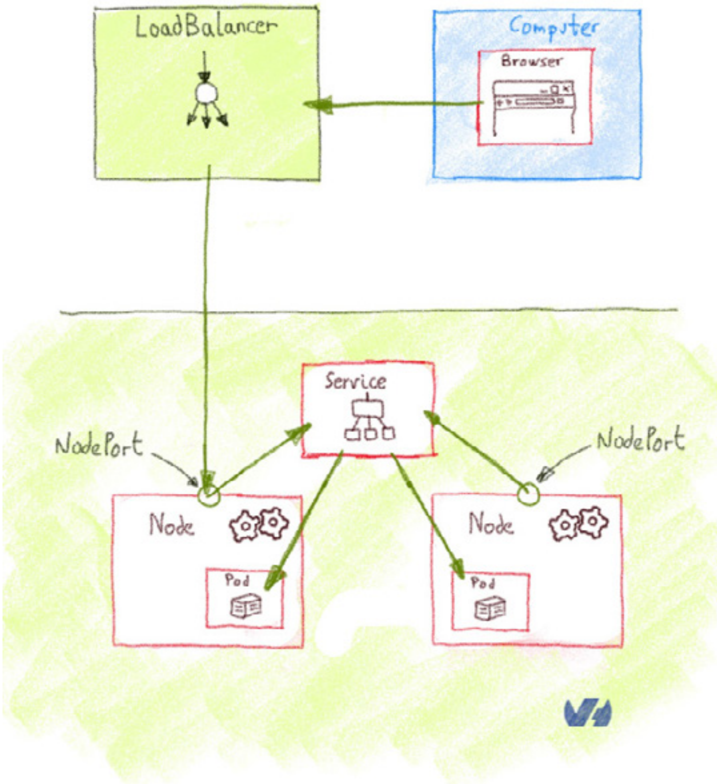
```
kubectl get svc hello-world-service -o wide
NAME                TYPE           CLUSTER-IP    EXTERNAL-IP    PORT(S)           AGE   SELECTOR
hello-world-service   LoadBalancer   10.3.86.116   141.95.96.63   8080:30219/TCP   18m   run=hello-world

➜ curl 141.95.96.63:8080
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
```

The URL of the node used in the curl command can be copied from the OVHcloud dashboard:



LoadBalancer: The NodePort service type is intended to be used for debugging purposes. To expose a service externally in a production environment we need a LoadBalancer service type. Creating a load balancer service inside the cluster triggers the creation of a public Cloud load balancer allowing public access to your application. On OVHcloud this service is charged hourly and will appear in the public Cloud project. Every load balancer on OVHcloud gets its own IP.



Example:
Similar to the example above with NodePort, the service type needs to be changed to LoadBalancer and that it is active verified (an external IP was assigned by OVHcloud):

```
kubectl get svc hello-world-service -o wide
NAME                TYPE           CLUSTER-IP    EXTERNAL-IP    PORT(S)          AGE   SELECTOR
hello-world-service   LoadBalancer   10.3.86.116   141.95.96.63   8080:30219/TCP   18m   run=hello-world

➜ curl 141.95.96.63:8080
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
```

Ingress resource: Another essential resource to define a single point of entry to a K8S cluster exists - an ingress, which is a separate resource and not a type of service (using Kubernetes' terminology). The ingress is used to define HTTP(S) routes to the services defined in a cluster, as shown in the picture below (3):

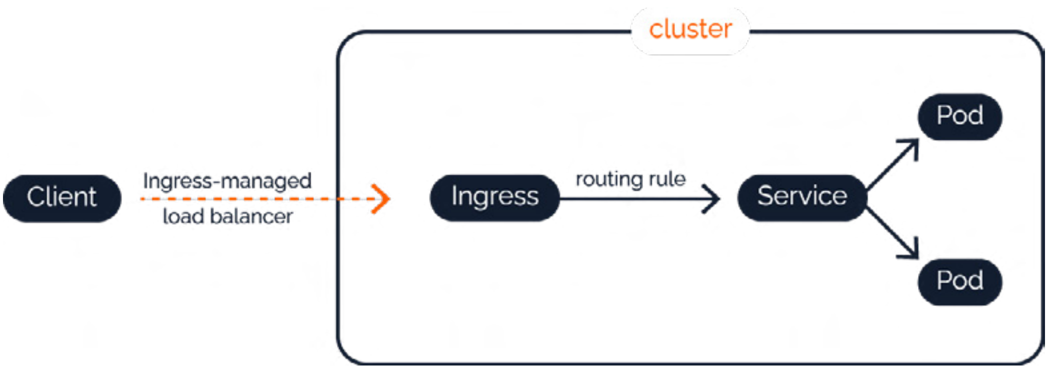


Figure 2 Ingress

Kubernetes on Kubernetes

Anyone who saw the movie Inception will relate to the approach taken by
OVHcloud when building OVHcloud Managed Kubernetes, i.e., kubeinception. Kubeinception means OVHcloud runs their customers' K8S clusters on their own K8S clusters (4), in other words, as Kubernetes over Kubernetes.
The main idea behind kubeinception is to run customer master nodes as pods in the admin cluster (OVHcloud's own cluster). To be more precise, the customer's master components are executed in OVHcloud's admin cluster nodes, namely: API Server, Scheduler and Controller (see picture below).

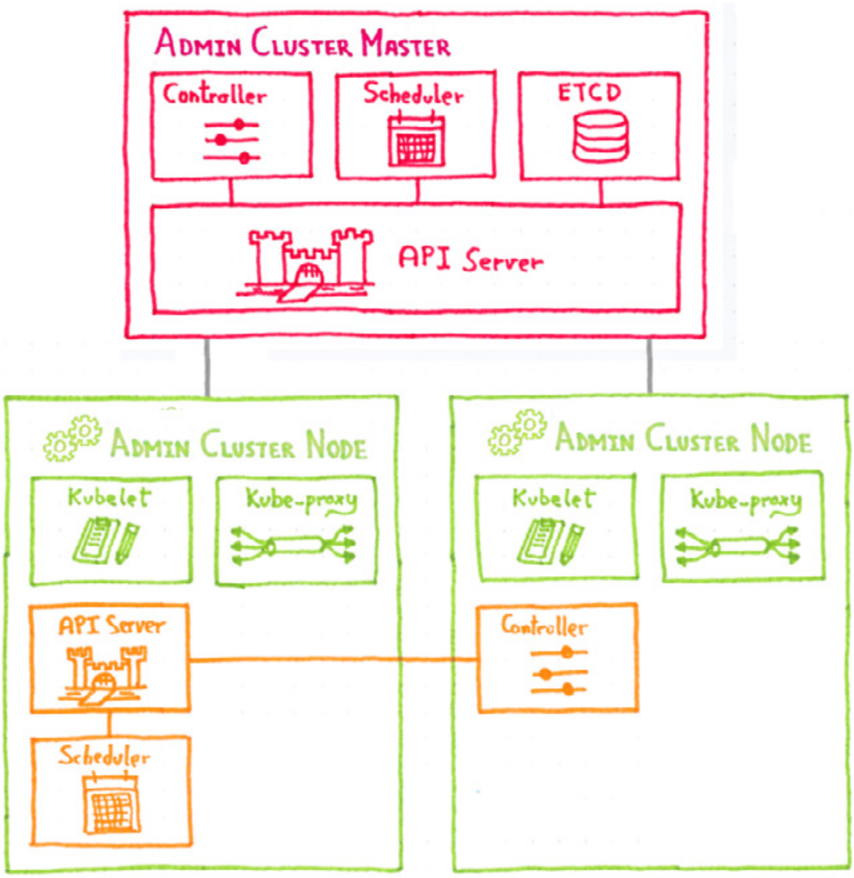


Figure 3 Admin cluster and customer master nodes

What about the customer's workloads? These are executed in normal Kubernetes nodes (OVHcloud public cloud instances connecting to API server running in the admin cluster), as shown below (setup with two customers, blue and orange):

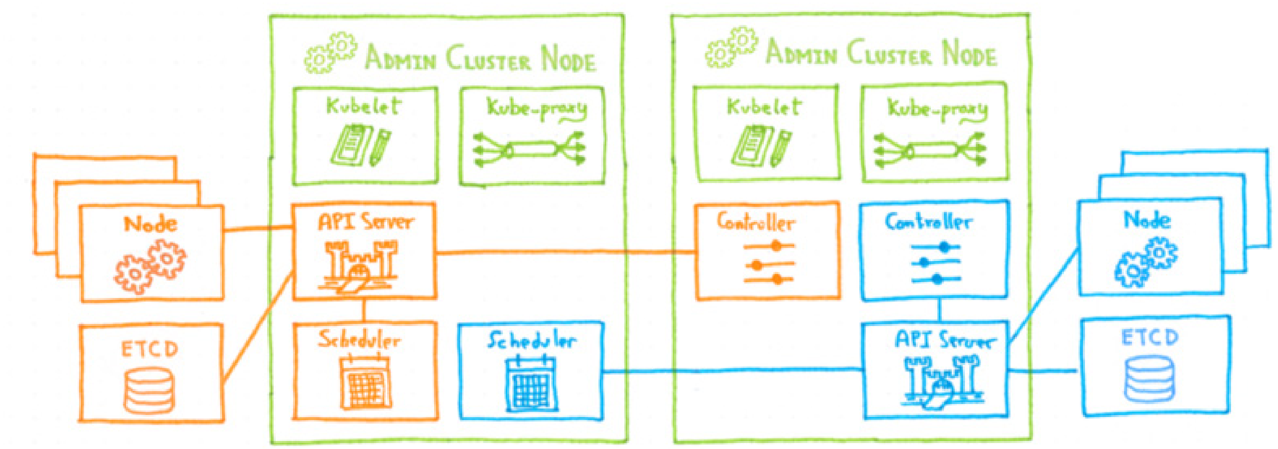


Figure 4 Admin cluster with two customer clusters deployed

The advantage of this setup is that customer master components are stateless (with ETCD externalized), and as 'vanilla pods' they are both as resilient as any pod running and are managed by Kubernetes (so will be restarted if a liveness probe fails).
Similarly, the worker nodes are vanilla K8S cluster nodes running outside the admin cluster; at that point they inherit the resilience of a normal Kubernetes cluster. Scaling is possible in both dimensions - the customer simply adds more node instances, while OVHcloud just deploys additional admin clusters.

What is PaaS in Cloud Computing ?

PaaS stands for Platform as a Service and represents one of the three cloud services models. PaaS sits between IaaS (Infrastructure as a Service) and SaaS (Software as Service) in terms of abstraction from the hardware. The primary target users of PaaS are developers (or small businesses) who want to start developing their application without the burden of operating the infrastructure that underpins it. Revisiting our Italian chef, PaaS is a ready-made kitchen with ovens, worktops and all the utensils needed to prepare a perfect Margherita - the chef just needs to come along with his ingredients and make the pizza! This might raise the question: where is Kubernetes managed in all of this? Here's a hint – vanilla Kubernetes is clearly below PaaS from an abstraction layer perspective as it allows you to build a PaaS.

Existing PaaS solutions

Some of the prominent implementations of PaaS are:

- Cloud Foundry
- OpenShift
- Google App Engine
- Heroku
- Force.com
- Apache Stratos
- AWS Elastic Beanstalk
- Azure App Service
- Plaform.SH

Differences between PaaS, IaaS and SaaS

The differences between On-Premises, IaaS, PaaS and SaaS can be explained with distribution of management overhead, even when they are built with the same software and hardware components. If everything is managed in-house, from server racks up to the Web UI of the application, this is On-Premises. If everything is managed by the cloud provider and a ready-made application such as Google Docs is used, this is SaaS.

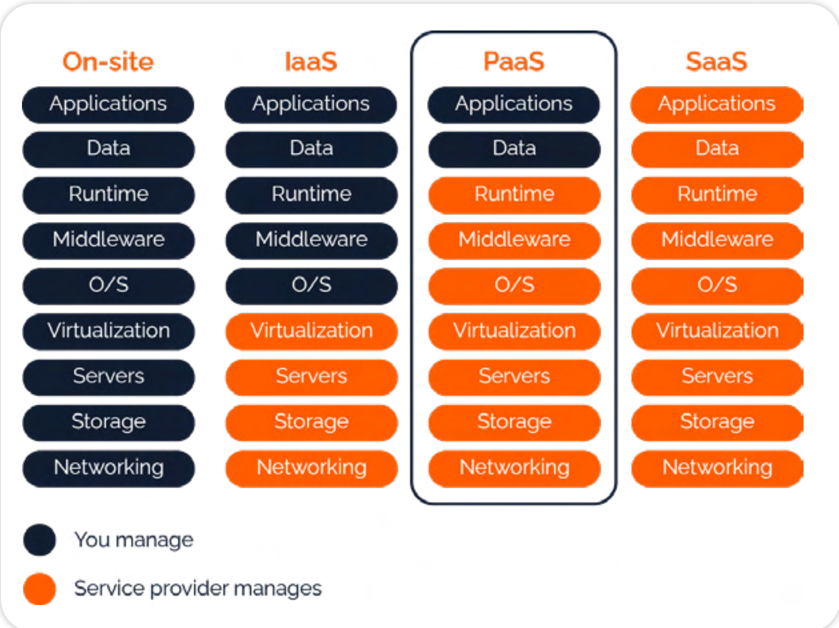


Figure 5 PaaS vs IaaS vs SaaS

In between these two extremes, one finds PaaS and IaaS. IaaS is all about computing instances (or virtual machines) which can be tuned to any purpose. At this point, estimating the resources current applications will actually need can become complex. PaaS provides a development environment including DevOps pipelines, BI services, storage services, and other middleware all running on top of the cloud provider's infrastructure.

Advantages and disadvantages of PaaS in Cloud Computing

The advantages of PaaS may be obvious: it is cheap and easy to use. The app is developed quickly and most cloud providers offer a free plan to get started on, providing an additional buffer before the monetisation model of a business kicks in. It would be reasonable to ask, where is the hitch? Our chef gets a perfect kitchen to make his Margherita pizza in, but what if he has another great idea that requires a slightly bigger oven? A developer might find themselves in a comparable situation when an application has special needs that are incompatible with the PaaS in place. For the sake of completeness, here are some more possible disadvantages of PaaS (5):

+ The codebase grows, making managing a basketful of services on the PaaS more and more difficult
+ A growing number of services, deployments etc. mean it is no longer cheap.
+ PaaS might not be supported in the particular region of interest
+ Our chef doesn't want a vendor lock-in (customer wants to go multi-cloud)
+ Specific needs of the application

Why choose PaaS

PaaS is still the best choice when an equation is to be optimized around the following factors:

+ Speed to market: especially interesting for start-ups that don't want to hire an army of DevOps before knowing that the business will succeed
+ Time savings: No time wasted on maintenance and setup of the stack
+ Cost effective: Pay-as-you-go is typically possible without needing to pay for idle resources
+ Future-proof: relying on state-of-the-art data centers and best practice, allowing users to scale out at any time

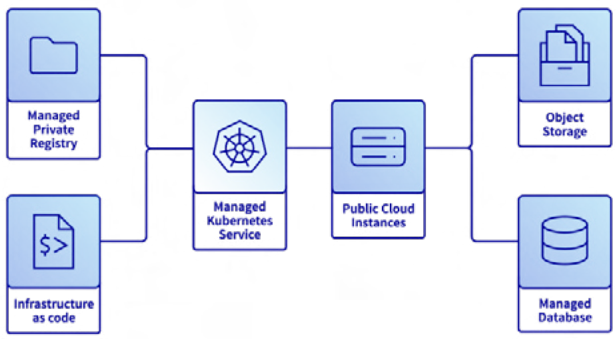
Kubernetes and PaaS

As noted above, vanilla Kubernetes is not PaaS, but using Managed Kubernetes as offered by OVHcloud moves it much closer to PaaS in several ways. OVHcloud's Kubernetes Service is fully managed (operational complexity is reduced to a minimum) and can be combined with several PaaS solutions e.g. a Managed MongoDB etc. bringing the quality of developer experience to a sweet spot.
The cherry on top? Users can rely on an Openstack based Public Cloud.

Openstack on Public cloud

OpenStack is an open-source platform for building and managing public Clouds, and is the foundation of the OVH Public Cloud offer. But why OpenStack at all? OpenStack is deeply integrated with Kubernetes and allows abstracting of underlying compute, storage, and networking resources using a standardized API (6).
The latter allows the exchange of public cloud instances with bare-metal servers also offered by OVHcloud. Bare-metal cloud instances provisioned by OVHcloud are not in the scope of this white paper, but here is a small teaser nonetheless: they allow even more optimized resource and cost usage!

Kubernetes is not enough



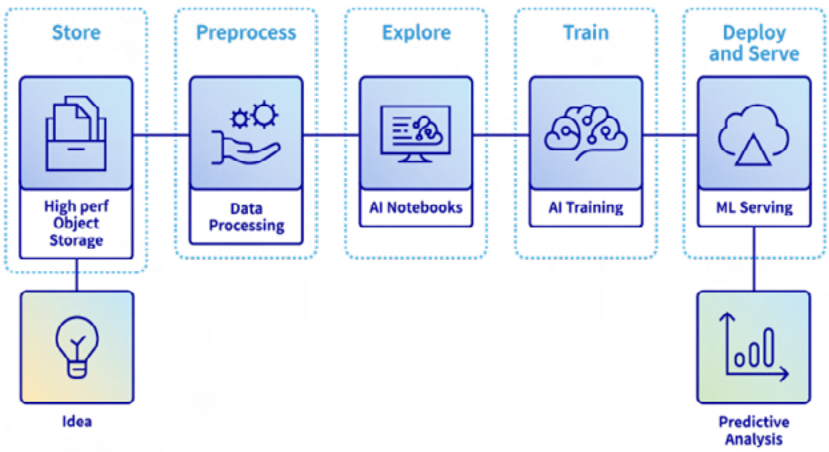
Some possible PaaS solutions that facilitate cloud-native strategies include:

- Managed Private Registry: use OVHcloud's private container registry to push container images securely and to pull from the K8S cluster during the deployment pipeline
- Infrastructure as code: use the OVHcloud provider on Terraform official registry to automate deployments of clusters and other cloud resources
- Object Storage: a store for static files such as videos, images and web files
- Managed Databases: managed database solutions from the most commonly used vendors (such as MongoDB, MySQL and many more). Using databases as a service allows the elimination of administrative tasks, which often end up being complex and are typically carried out by a Database Administrator (DBA).

AI Pipelines

Further PaaS solutions exist that can assist in getting the most out of your data by using artificial intelligence (AI) and setting up a data pipeline. The latter commonly involves the following steps:

- Store: use a data lake to store data
- Preprocess: use a data processing solution powered by Apache Spark
- Explore: use AI Notebook to launch a Jupyter or VS Code with the most popular AI frameworks installed, simplifying the use of data
- Train: Train the AI model using the OVHcloud AI Training solution (built on the Kubernetes platform and running on CPU/GPU nodes)
- Deploy: ML Serving solution enables serving the model from the previous step using a de-facto industry format.



Improving the DevOps experience

This section outlines all the benefits of using OVHcloud Managed Kubernetes in conjunction with other PaaS solutions. These are all about improving the DevOps experience by increasing automation, and improving upgrade and rollback processes, as well as security and availability.

CI/CD pipelines

To support the CI/CD process OVHcloud offers the Managed Private Registry, built on top of open-source solutions considered the industry standard - namely, CNCF Harbor supporting OCI images and HELM charts which in turn streamlines deployment to K8S clusters. The use of pipelines to trigger automated tests, image building, and deployment of a new version of an application can also be achieved and automated using pipelines in another OVHcloud solution called CDS (Continuous delivery service). Such a pipeline consists of several stages, each stage representing one or several jobs. Typically, one stage might focus on CI (Continuous Integration) which includes testing, building, and pushing an application image to the registry, with a second stage making sure the new application version ends up in the Managed Kubernetes Cluster.

Sound Architecture designed for scale

As explained earlier, the Managed Kubernetes Architecture is built on a sound base: OpenStack and kubeinception. CNCF certified, it is compatible with any vanilla Kubernetes solution and prevents vendor-lock, offering scalability on the one hand, and on the other, reduced Ops complexity. Furthermore (as also noted previously) it is not only the application deployment process that can be automated, but the rollout of environments (or clu-sters in which containerised applications are running) can also be highly automated using Terraform.

Security in the cloud

In DevOps, security is not an afterthought but part of the entire lifecycle. The OVHcloud team makes sure every Kubernetes cluster receives updates and security patches. Even the hardware used to run the cluster benefits from OVHcloud's Anti-DDoS protection. A private virtual network adds another layer of security; this is achieved through integrating the Managed Kubernetes service with vRack (used to link different OVHcloud services to each other using network peering). Role-based access control is available by default and can be used to deploy applications with distinct levels of access or to limit the actions a user can perform in a cluster. The process of pushing container images to OVHcloud Managed Private registry includes default checks for vulnerability as well as for images' source integrity.

Bibliography

01. Ingress. kubernetes.io. [Online] 20 03 2022.

02. Yann Irbah. Kubernetes and the Next Generation of PaaS. THENEWSTACK. [Online] Qovery, 09 02 2022.

03. Ramon Acedo, Anita Tragler, Franck Baudin. Run Your Kubernetes Cluster on OpenStack in Production. superuser. [Online] superuser, 16 March 2021.

04. Powell, . Do I Really Need Kubernetes? . thenewstack. [Online] 5 October 2020.

05. Tammy Butow. How to Create a Kubernetes Cluster on Ubuntu 16.04 with kubeadm and Weave Net. Gremlin. [Online] 27 02 2018.

06. Horacio Gonzalez . Kubinception: using Kubernetes to run Kubernetes . OVHcloud Blog. [Online] OVHcloud, 25 01 2019.