1. Cloud Platform

Content

2. Architecture Pattern

2.1. ACID

2.1.1. Step 1

- Docker EE K8S requirements:

  - Create AD Group and add application team members
  - Create K8S namespace (with ACID assistance),
  - Setup K8S environment.
- OpenShift Requirements:
  - Give access for ACID Team to namespace OpenShift.
- Build requirements:
  - Create Docker repository.
- Operation requirements:
  - Present hld & define a SPoC
  - Jenkins provisionning
  - Provide flow matrix

2.1.2. Step 2

- Docker EE K8S requirements:
  - Create basic docker images.
  - Get dependencies (npm, yarn, jdk, python…)

2.1.3. Step 3

- Generate ACID template

2.1.4. Step 4

- Deploy application on K8S:
  - Adapt templates and configurations.
  - Connect to third party applications and services (DBs, Storages…)

2.1.5. Go Live

- ACID Team:
  - Test application (healthcheck, access, …)
  - Send Pull request.
- Application Team:
  - Validate Pull request.
  - Re0use ACID template for ENV (hom, prd, …)
  - If OK, decommissioning Openshift environments.

2.2. Twelve-Factor Applications

- Codebase
  - version control system
  - codebase isolation per application
  - invoicing of reusable code transverally in another codebase
- Dependencies
  - Explicitly declare isolate dependencies
- Configuration

    - Separate code from configuration
- Backing Services
    - Treat backing services as attached resources
- Build, Release, Run
    - Strictly separate build and run stages
- Processes
    - Execute app as stateless process and don't share anything.
    - stateless application allows implmentation of horizontal scalability (multiplication of processes executing the same task to absorb a peak load) and simplifies operations in Prod
- Port Binding
    - Export services via port binding
    - Routing layer manages the routing of requests from a host name, which is exposued to users to the processes on which the port is associated.
    - Advantage is to limit the impact on users in case of re-urbanization of the service.
- Concurrency
    - Scale out via the process model
- Disposability
    - Maximize robustness & graceful shutdown
- Dev/Prod Parity
    - Keep dev, staging, prod as similar as possible
- Logs
    - Treat logs as event stream
- Admin Processes
    - Run admin/mgmt tasks as one-off processes

Comments:

- reduce time and cost of implementing
- provide max portability between runtime environments
- decoupled from infra to reduce administration tasks for underlying systems and improve fault tolerance
- minimize gap between dev and prod, allow continuous deployment for max agility
- can grow (scalability) without significant change intools, architecture or development.

2.3. Infrastructure as Code (IAC)

IAC is managing and provisioning infrastructure through **software** to achieve **consistent** and **predictable** environments.

Benifits

- Automated deployment
- Consistent enviroments

- Repeatable process
- Reusable components
- Versioning code and infra
- The build of immutable & repeatable infrastructures
- The unit testing of the infra
- The build of many environments quickly
- The scalability and resiliency of infra
- Cost optimizations

Programmable infrastructure = management and provisioning of infrastructures throught the code (Infra as Code) was implemented as an enabler of CI/CD philosophy.

IAC is intelligent enough to calculate dependencies and create the resource in the correct order. (Ex: Terraform)

2.4. Deployment Patterns

- Blue Green
  - Technical release that reduce downtime and risk, runs 2 identical production environments (blue & green), with only 1 environment live, serving all production traffic.
- Canary Release
  - Reduce the risk of introducing new software version in production by slowly rolling out the changes to small subset of users before rolling to all users
- A/B testing (Split Testing)
  - Define the most effective version of 2 implementation of a feature
- Rolling update
  - 0 downtime of microservices. One instance running n+1 version is created before one instance running n version is destroyed.

2.5. Architecture Patterns of Cloud Native principles

2.5.1. Decoupling - Architecture Pattern

- Objective
  - Application services are not coupled with the underlying infra to facilitate operations.
- Best Practices
  - Applications are hosted on standardized virtual infra and don't use specific infra capabilities.
  - Applications consume and expose services that are accessible and documented by APIs.
  - Services are stateless.
  - Standard protocols are used to invoke synchoronous or asynchronous calls.

    - Use containers to be completely independent of infra (Docker).

2.5.2. Automation - Infra as Code

Infra as Code automates deployments and avoid manual configuration, from infra layer to software layer using tools like Terraform, Ansible or Puppet.

Infra as Code is integrated into CI-Continuous Integration and CD-Continous Deployment pipelines to mount and destroy resources as soon as pplication source code is updated.

- CI process: assembly and unit testing, both at infra and application level.
- CD process: automation of prod release integrtating application and infra scale-up testing

Infra as Code is a component of DevOps business model.

2.5.3. Elasticity (Scalability) - Architecture Pattern

- Vertical elasticity (scale-up): adding capacity to already provisioned resources. It's complex to add or remove resources without interruption of service.
- Horizontal elasticity (scale-out and scale-in): duplicate resources identically, to meet scale-up needs. Application must meed most of the 12 factors.
- Objective
    - applications respond to demand levels, growing and shrinking as required, in and among Clouds.
- Pattern
    - **Stateless** services to facilitate the **horizontal** elasticity of an application.
    - Each instance creation/deletion must be declared in an upstream load balancer service to guarantee accessibility of each available instance without requiring any change from application users.
    - When services need to store information related to user session (frontend), we use **external storage** resource to store the user session and preserve the user session outside the context of the service.

2.5.4. Failure Awareness - Architecture Pattern - Database service

Pattern: use replication of storage resources for persistent application data.

2.5.5. Security - Architecture principles

- Access control
    - Datacenter

        * All Internet flows are filtered via Internet Access Point (IAP) and managed in dedicated areas (web cells).
        * All flows exchanged with a public cloud are filtered via a secure access point (Cloud Bridge) and managed in dedicated areas (cloud cells).
    - Platform public cloud
        * An Internet access point is hosted in a Virtual Private Cloud (VPC) used to secure all Internet flows.
        * A dedicated VPC to administrators is used to secure and track administration operations.
- Confidentiality
    - VM images are encrypted in public clou.d
    - Encrypted VM images are managed by orchestrator hosted in VPC.

2.6. Example of Cloud Native Architecture Design with Infra as Code

- Decoupling to meet the challenges
    - security
    - autonomy of funtionalities
- Dynamic elasticity to meet challenges
    - High availability
    - Load absorption in compliance with SLAs
    - Management of technical debt
    - Run cost optimization
- Deployment of Infra as Code via pipeline to perform resource creations.
- Include **self-healing mechanisms** that can be triggered by the application or infra through partial or total deployment.

3. Cloud Platform

3.1. Terminology

- SLA (Service Level of Agreement)
    - commitment between service provider and service user
    - based on quality, availability and responsibilities
- SLO (Service Level Objective)
    - set of measurable values that characterize a service reliability and upon which an SLA is based
    - Cloud platform service SLO is based on the values of RTO, RPO, Availability and type of failures (local or regional).
- SLI (Service Level Indicator)
    - set of measures that evaluates whether system has been running within SLO.
- RTO (Recovery Time Objective)

  - targeted duration of time within which a service must be restored after a disaster/disruption to avoid unacceptable consequences associated with a break in **business continuity**.
  - actual data loss before incident
- RPO (Recovery Point Objective)
  - max targeted period in which data (transactions) might be lost from an IT service due to a major indicent
  - actual recovery period after incident
- AZ (Availability Zone)
  - set of physical data center(s) in a region that hosts infra resources provided by cloud platform.
  - Each zone in a region has redundant and separate power, networking and connectivity to reduce likelihood of 2 zones failing simultaneously.
  - AZ within the same region are sufficiently close to be interconnected with large bandwidth and low latency network links which allow synchronous data replication.
  - AZ sufficiently far can have a single domain failure.
  - Some IT dept have regulatory contraints that require DR plan in second region.
- Loadbalancer
  - dedicated virtual machine that handles the loadbalancing activity (redundant or not)
- VIP
  - An IP address defining an entry point to the application
- Subnet
  - IP range to which the VIP belongs
- Listener
  - service on which the loadbalancer accepts incoming application traffic
- Pool
  - group of virtual machines to which the loadbalancer distributes the application traffic
- Health-check
  - probe used by the loadbalancer to check if a pool member is ready or not to receive application traffic
- SSL profile
  - object defining protocols and the algorithms used to secure communications
- Swagger
  - open source software for designing, documenting and consuming RESTfull web services.

3.2. XaaS Model

- XaaS = Everything as a Service. All cloud platform services are built as XaaS.

- Control Plane (CP) of a cloud platform service exposesan interface through which customers provision and then manage resources. CP is responsible for managing lifecycle of resources.
- Resource Plane (RP) is a set of resources that are managed by control plane (CP) and supports business applications and thus are business critical components.
- Cloud Platform services have SLOs differentiated by CP and RP.

3.2.1. Infra as a Service (IaaS)

- lowest level of service
- providing access to a virtualized computer park. Customer can isntall OS and applications on VMs, similar to traditional data center hosting services
- Trend is towards higher level services (PaaS, SaaS)

3.2.2. Platform as a Service (PaaS)

- OS and infra tools are the responsibility of the cloud provider. Customer has direct access to specialized services (frontend, middlewares, database) that can be customized: customer chooses, in a marketplace, the design of tech solutions to be deployed. Cloud provider execute the implementation with its own scripts.
- Advantage
  - customer of cloud solution can focus on the business application to be implemented
- Disadvantage
  - PaaS provider decides to no longer authorize the use of a tool or version of the tool, or interrupt or develop its offer, consumer must adpat within timeframe chosen by PaaS provider.

3.2.3. Software as a Service (SaaS)

- software available to clients, accessible from Web browser, customer doens't worry about making updates, adding security patches and ensuring availability of service.
- Advantages
  - Rapid implementation and provision of services for customer. No worry about software packages and licenses anymore. Service immediately available online.
  - As software is managed by provider, customers no worry about deployments, maintenance and tech evolution phases of service.
- Disadvantages
  - difficult to manage service interruptions: customer have to organize themselves around the constraints imposed by SaaS provider
  - necessary to comply with supplier's prerequisites for a good use

    - customer is subject to SaaS product lifecycle by cloud provider, particularly in the event of a lack of functionality
- SaaS provider can operate PaaS-type services, which in turn can use IaaS

3.2.4. Summary

- IaaS host Servers.
- PaaS build Servers and Middleware/OS.
- SaaS consume Servers, Middleware/OS and Applications.

3.3. Public Cloud Private Cloud

3.3.1. Public Cloud

- Infra is mutualized and is shared between several companies and individuals.
- SLA (Service Level Agreement) is the same for all users and defiend by provider.
- Public Cloud services are provided in virtualized environment, built using physical resources in pools shared with all other subscribers.

Benefits:

- High resource scalability so that an app can respond flexibly to charge fluctuations.
- Pricing model is based on what is consumed, no more need to tie up resources to anticipate future needs
- High availability of access to environments.

3.3.2. Private Cloud

- exclusively used by a company
- hosted in company-owned datacenter

3.3.3. Hybrid Cloud

- Technological standards to communicate between clouds.
- Network connectivity that meet the needs.
- Can shift workloads between different types of hosing according to changing needs, cost recution or in case of a need for dynamic increase of mass resources.

3.4. Traffic Manager and Service Load Balancing

- Traffic Manager
    - DNS resolver with advenced routing capabilities
    - global service that ensures inter-AZ resiliency
    - could be requested by Corporate DNS using friendly host aliases

  - Routing Capabilities
    - waited for active routing (recommended for multi-AZ deployments)
    - By priority for active / standby routing (recomended for multi-region deployments)
  - Health check support
  - Members: VIP only
- Service Load Balancing
  - load balancing L4/L7 service that supports TCP/HTTP/HTTPS protocols
  - zonal service with a built-in intra-AZ resiliency
  - Persistence (stickness): client IP (useless behind proxy/sipo), cookie based (for HTTPS, perform SSL termination at the SLB level)
  - IP Source visibility (XFF)
  - Health check support
  - Backend Services: OCS/VCS computers, Cloud compute (useful for seamless migration)

4. Cloud Platform Services Resiliency Pattern

4.1. RabbitMQ Messaging Pattern

- Multi AZ highly available RabbitMQ brokers using queue mirroring
  - Use this pattern for applications that need multi-AZ messaging persistence
  - Deploy a stretched RabbitMQ cluster over 2 AZ within the same region using Queue Mirroring, but only with 2 nodes.
  - When split brain occurs, you won't have message loss. Clients are connected at only 1 node at the same time. Moreover, split brains are monitored with automatic healing.
  - In case of AZ outage, the failover of a RabbitMQ cluster is automatic and seamless for the users as well as the switch to the nominal state.
- Multi Region resilient brokers patter
  - Use this pattern for applications that need multi-region messaging resiliency.
  - Create a RabbitMQ cluster with at least 2 nodes per AZ with queue mirroring (local HA)
  - Use RabbitMQ Exchange federation between 2 clusters whenever you want to ensure multi-rejion persistence.
  - RabbitMQ Federation is WAN-friendly asynchronous message replication
  - Since message replication is asynchronous, AZ/region outage could lead to few message loss.
  - For a seamless automatic failover, your consumer application must be idempotent.

4.2. Resiliency with Most Common Application Patterns

- Use multi-AZ Active / Active Traffic Manager/SLB/OCS compute pattern
- Frontends at each AZ will point the PostgreSQL Master node at nominal state. These components will point to the replica node for failover in case of Master failure.
- Database failover when the Master is down is manually triggered by the project team (API call)
- Ensure your application components will point to the new master node when database failover is triggered. Project team create a new replica as soon as possible. For Edge provider, the new replica has a few fqdn/ip and hence you need to update your client datasources.
- Use RabbitMQ multi-AZ pattern with Queue Mirroring if you need regional persistence of messages. Use Local HA RabbitMQ cluster in each AZ if you need a zonal persistence of your messages.
- RabbitMQ Failover and switchover are automatic.

Onyx's high-level architecture

4.3. Container Hosting

- Orchestrated Containers is based on Docker EE solution with Kubernetes as a container orchestrator
- Orchestrated Containers offer delivers Kubernetes Namespaces as a Service (CaaS). Each namespace is created in client zone.
- Namespaces are regional resources: Kubernetes clusters are stretched across 2 AZs within a region to simplify multi-AZ application deployments and cross-AZ application traffic.
- Each Namespace has its own multi-AZ ingress connectivity based on the folling components: Wild Card DNS/Traffic Manager/SLB/Ingress Controller
- Kubernetes Managers are based on distributed consensus clustering. Regions have at most 2 AZ's which leads to odd distribution of the Managers. Manual operations are needed to recover Kubernetes cluster in case of AZ failure (case where the failing AZ has the majority).
- AZ-aware deployments (using AZ affinity based on worker labels) are not impacted in case of AZ failure.

4.4. Multi-AZ pattern for Stateless Components without AZ Affinity

- Use Multi-AZ deployment without affinity for simple deployments and let Kubernetes handle the location of pods where the resources are available
- This pattern is suitable for stateless applications using Kubernetes **Deployment** controller
- In case of AZ failure, Kubernetes will automatically reschedule pods in the available AZ when the cluster is back again after recovering operations.

- Use an Ingress rule only if you need to expose your application service outside Kubernetes cluster.
- By default, prefer a TLS passthrough ingress rule with virtualhost sni-based routing. This ensures an end-to-end TLS traffic.
- Set a TLS termination at the ingress controller level if you
  - need sticky session or client IP load balancing mode.
  - need a path-based routing.

4.5. Muilti-AZ pattern for Stateless Components using AZ Affinity

- Use Multi-AZ deployment with AZ affinity for critical deployments.
- This pattern is suitable for stateless applications using Kubernetes **Deployment** controller
- Can set up a pod affinity with a **topologyKey** based on the AZ lable of Kubernetes nodes (workers).
- From Kubernetes >= v.16, can spread evenly your pods across multiple AZ more efficiently using the feature **topologySpreadConstraints**.
- In case of AZ failure, your scheduled pods will continue to run but no new pds can be scheduled until recover manually the cluster. If running pods are well sized, your business application will continue to run.
- For the ingress rules and TLS implementation, use the same recommendations described in the previous pattern.

5. Containers and VM

5.1. Containers

- Containers are isolated, but share OS and, where properly, bins/libraries.
- Result is significantly faster deployment, much less overhead, easier migration, faster restart.

Benefits:

- Agility
  - fully manage deployments
- Autonomy
  - manage everything, from application topology to load-balancing and DNS wildcard.
- Scalability
  - A rush of clients? Simply scale in or out.

5.2. Orchestrated Containers

- Environment where you can run Container Image. An image is the smallest artifact on container world.

- An orchestrator (Kubernetes), in charge of launching your containers on compute resources, with the number you requested.
- Internal networking capabilities, to make container communicating with each others
- Network chain which allows the access to your hosted application from corporate network.

Do on Orchestrated Container Platform:

- Host small, microserviced parts of applications on Orchestrated Containers service
- Host open-source software
- Host stateless services
- Host service with permanent small-traffic, and autoscale for peak days (end of month, yearly intensive activity).

5.3. Orchestrated containers use OpenStack as backend

- Infrastructure Platform (IaaS)
  - AWS EC2, Azure VMs
- Container Platform (CaaS)
- Application Platform (PaaS)
- Function Platform (FaaS)
- Software Platform (SaaS)
  - Oracle, Salesforce, SAP

5.4. Container vs. VM

Sharing (containers) some underlying OS, or not (VM, each have its own).

Type	Container	VM (OCS)
Isolation	Shared OS with logical isolation based on linux kernel cgroups and namespaces	Full isolation of OS
Operating System	Only executing user mode of OS. Less consuming and customizable	Full OS, consuming more resources
Lifecycle OS (patch)	Rebuild images and redeploy containers	Rebuild images and redeploy VM
Storage	Local storage, block storage, object storage	Local vm storage, block storage, file storage for shared data or object storage
Workload balancing	Managed by Kubernetes	Managed by OpenStack

Type	Container	VM (OCS)
Fault Tolerance	Self healing managed by Kubernetes	Need to setup automation using monitoring
Network	Container: overlay network (dedicated to K8S cluster) and micro segmentation managed by K8S using network policies	Use virtual NIC

5.5. When to use VM

- Need different OS than those supported by the orchestrated containers registry service due to software editor prerequisite. Windows containers for instance, are not supported
- Want to lift and shift and doesn't have time to transform applications
- Need support for virtual appliance, and editor doesn't support Containers.
- Licensing doesn't fit container pricing model (Oracle procuts)
- Need an egress fixed IP for route opening or proxy whitelisting (ProxyApp or SMTP relay, access to L3 network zone services)
- Need to filter by source ip at ingress level

5.6. When to use Containers

Everything else.

5.7. Containers Use Cases

- Containers are best fitted to host web application stateless frontend components
  - Ex: Nginx, Apache, Spring Boot
- Deploy stateful applications with Orchestrated Containers service using persistent volume based on block storage
  - Ex: NoSQL databse, KV database, messaging components like Kafka
  - **Notes:**
    - Be careful when using persistent volumns, don't deploy standalone components (volumes aren't replicated cross AZs)
    - To achieve multi-AZ resiliency, stateful applications must be deployed across all AZs using TopologySpreadContraint
    - Persistent volumes are ReadWriteOnce (can be mounted on one pod only), the pod and the volume mounted have to be in the same AZ
- Scheduled and on-demand batch Workloads using Kubernetes CronJob/Job deployments

- **Benefits:**
    - Simplified infra provisioning: Create a Kubernetes Namespace and that's it, no VMs, no SLB, no Traffic Manager
    - Kubernetes built-in functionalities: Self-healing (Liveness probes), Auto-scaling, rolling updates (ZDD), Fast startup
    - Costs: resources consolidation, time-based scaling possible to reduce non-production environment billing

5.8. Cloud Native

5.8.1. Definition of Cloud Native Application

- distributed, elastic and scalable system
- various services
  - application core, storage, network, authentication, monitoring, orchestration, etc.
  - integrate small components carrying a single process (microservice) - autonomous, stateless and weakly coupled - designed to be scalable.
- Microservices communicate with each other being agnostic of the languages used.
- Objective: use underlying resources as economically as possible
- Deployment of Cloud Native Application is automated, at each stage, and performs technical and functional tests before going into prod (CI/CD pipeline)

5.8.2. Key Characteristics of Cloud Native Application

- Infra independent
- Location independent
- Resilient to failure
- Service-oriented architecture (SOA)
  - Applications consume and expose web services with APIs discoverable at runtime. The structure incorporates small, stateless components designed to scale out.
  - Services are loosely coupled.
  - All functionality/services are published and consumed via web services (API).
- Elastically scalable
- Designed for manageability
- Cost & resource consumption aware
- Bandwidth aware
- Secure

Principles of above:

- API programming
  - Cloud Native Application exposes the API of each of its services.

    - Calling a given service API triggers asynchronous communications between its microservices which perform tasks.
        * Decoupled services exposing APIOs and communicating by messages. In case of corruption or loss, only the microservice has to be redeployed.
- Cloud Native Architecture design pattern = 12-Factor APP
- Microservice-oriented architecture (a service should do one thing well)
- Stateless services
- Infra as Code

Whether you're using VMs or Containers, note that:

- build Docker images are faster than build OSF images
- containers deployment are faster

5.8.3. Micro-Services

Micro-services oriented architecture is

- Evolving
    - Deployment of funtionalities without downtime via CI/CD and versioning management
    - Interoperability of different technologies or development languages in the same application
- Modular
    - Each functionality is autonomous, independent, resilient, scalable and elastic
- Agile
    - Application lifecycle by functionality
    - Lightweight communication mechanisms in asynchronous mode (API)

To build application based on microservices:

- Input
    - Stable interface contract must be published
    - This allows to be autonomous and independent to develop and bring evolutions
- Output
    - Formats must be published so that the next function can evolve in total autonomy
    - There can't be processing without output

Application:

- More distributed and dynamic systems
- Need to move from host-centric to service-centric observability.

5.8.4. Component Type: Stateful vs. Stateless

- Stateful component
  - stores certain data that will be required for further processing
  - persistence area is stored locally
  - Advantages:
    * As data stored locally, processing is faster than with a stateless architecture.
  - Disadvantages:
    * In case of interruption of a session, you must have implemented audit trails of fines to carry out recovery.
    * If component is lost, data is lost. Often complex mechanisms must be implemented to overcome this.
    * If component is duplicated to bring power, it will be necessary to implement data sharing solutions to ensure consistency.
- Stateless component
  - doesn't sore any data that needs to be kept longer than transaction time
  - Processing of transaction will follow this scenario:
    * Identification of info issued by customer
    * Retrieving additional info in an area external to the component, the persistence area (often a database)
    * Performing processing
    * Provision of result for further processing
    * Contect cleaning (especially for data confidentiality)
  - Advantages:
    * Each transaction is independent, no link with previous or next transaction. Good elasticity of these components without prioir modification.
  - Disadvantages:
    * Treatments are longer than stateful architecture because there can be consultation of external info.

5.8.5. Cloud Native architecture vs. Traditional architecture

Domain	Traditional IT Architecture	Cloud Native Architecture
Hosting & Infra	Physical and virtual infra	Virtual infra, no adhesion between application and infra
Application Type	Monolithic Application or N-Tiers, stateful	Microservices applications, stateless
Exchange between applications	Synchronous, Asynchronous	Asynchronous via API

Domain	Traditional IT Architecture	Cloud Native Architecture
Components	Specific components for each application	Standardized components for all applications via a common repository
Resiliency	Managed at infra level	Managed at application level (auto-healing, auto-scaling)
Quality	Tests with low levels of automation	Automated tests via pipeline CI/CD
Monitoring	Logs generated locally by each server of the application	Metrics collected and consolidated by a common service external to be the application
Operating Model	Separation of the Dev and infra teams	Infra as Code DevOps/NoOps/FinOps

5.9. Cloud Platform Availability

Availability of a system with n redundant services having an availability of $A\%$ each, is

Availability $= (1 - (1 - A/100)^n)$

Availability of a system with n dependent services having an availability of $A_1, A_2, ..., A_n$ each, is

Availability $= (A_1/100) * (A_2/100) * ... * (A_n/100)$

Ex: Load Balancer = 99.9%, going to two redundent web servers with each 99.5%, then going to database = 99.5%, the availability of the system $= 0.999 * (1 - (1 - 0.995)^2) * 0.995 = 0.994$.

6. Cloud Platform Architecture: Kubernetes and Terraform

6.1. Terminology

- Docker container image
  - A lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings.
- Container

  - An instance of a Docker image running as an isolated process in a user space of an OS Server.
- Pod
  - A set of one or more containers sharing network namespace and local volumes
  - Each Pod has an IP address in Kubernetes cluster's private network (aka overlay network)
  - Containers within a Pod communicates through localhost interface
  - Containers of a Pod are collocated in the same worker node.

6.2. Orchestrated Containers

- Kubernetes is a popular open-source container orchestrator for automating application containers deployments and offering features like scaling, load balancing, service discovery, rolling updates, auto restarting/healing, etc.
- Docker Enterprise Edition (Docker EE), is an integrated container platform that supports both Swarm and Kubernetes orchestrators. It comes with out-of-the-box features like multi-tenancy management, networking, enterprise private registry and advenced security features.
- Orchestrated Containers service is based on the Docker EE solution with Kubernetes as a strategic container orchestrator.

6.3. Deploy stateless components of an application on Kubernetes Cluster using Cloud Platform Orchestrated Containers service

- Kubernetes Cluster
  - Multiple Manager nodes are deployed to ensure high availability of the Kubernetes cluster
  - Multiple registry nodes are deployed to ensure high availability of the built Docker images
  - Multiple Kubernetes nodes (aka workers) are deployed in the clients network zones to handle business applications pods/containers workload
  - All API operations of Kubernetes clusters are handled through `kube-apiserver` component, which controls the state of the cluster: nodes, endpoints, replicas, service account.
  - `kube-scheduler` is responsible for workload management: schedules Pods on nodes with the requested resources (CPU, RAM).
  - `etcd`: Distributed key-value database that stores all Kubernetes cluster's data
  - `Kubelet`: gets Pods configuration from kube-apiserver and ensures that all containers are in the expected status
  - `Client Namespace`: provides a way to divide cluster's resource for multiple users and ensure resource isolation.
- Deliver model for best user experience

    - Serverless → fully managed clusters
    - It's a Kubernetes Namespace as a Service (CaaS) delivery model using onboarding API
    - Create Kubernetes Namespace and the external network connectivity (Traffic Manager/Load Balancing) with just 1 API call
    - Keep focus on your application transformation

6.4. Common Architecture Pattern

- Frontend App
    - Stateless components fitting containers hosting
    - Cloud Platform = Orchestrated Containers Service
- Backend (Business data replication)
    - Stateful components fitting database hosting
    - Cloud Platform = Postgresql Service

6.5. Deploy secrets on Kubernetes

- Create and store secret files of business application in a safe place like MyVault
- Load secrets on Kubernetes
- Secret registered with success

Notes:

- Don't store secretes (private keys, passwords) in Docker container images or other public deployment manifest
- Store secrets using Kubernetes Secret objects. Secrets will be available only for your Namespace and could be mounted on your containers as a tmpfs volume or loaded as environment variables.

6.6. Application Deployment on Kubernetes Cluster

- Deploy Application secrets and configuration
- Deploy 4 pod's replicas of the stateless components using the Application build image
- Deploy Kubternetes service with a ClusterIP and RoundRobin Load balancing between 4 pods
- Deploy Kubernetes ingress rule to expose your application service
- Access your application from browser

6.7. Terraform Deployment

6.7.1. Terraform

- open-source infra as code software tool, can automatically build, modify and delete an infra.

- Doesn’t create resources in the sequential order of the code, but according to dependency between resources.
- Parallizes the creation of independent resources.
- Dependency analysis is done during calculation of the execution plan.
- Use declarative approach and keeps a list of current state of system objects.

Benefits:

- Manage immutable infra
- Declarative, not procedural code
- Client-Only architecture
- Multi-platform / portability
  - One tool and one language for describing infra for Google Cloud, AWS, OpenStack and so on clouds.
- Large community.

6.7.2. Terraform Components

- HCL (HashiCorp Configuration Language)
  - Configuration files are written in HCL. When executing, Terraform scans all files ending in `.tf` or `.tfvars`
- Provider
  - Provider is responsible for lifecycle/CRUD of a resource
- State
  - Terraform store state about managed infra and configuration by default in a local file named `terraform.tfstate`. It can be stored remotely, which works better in a team environment.
  - Terraform uses local state to create plans and make changes to infra. Prior to any operation, Terraform does a refresh to update the state with the real infra.
- CLI
  - `init`: load provider and modules
  - `plan`: create execution plan
  - `apply`: apply changes
  - `destroy`
  - `show`
  - `import`
  - `state`

6.7.3. Deploy outcome with Terraform

- a real stateful application (an open source Slack)
- with secret management
- on 2 AZ in one region

6.7.4. Architecture

- High Availability

  - Traffic Manager → high availability between AZs
  - Load Balancer → high availability in AZs
- Application Tier
  - Stateless → horizontal scalability
- Data Tier
  - PostgreSQL: posts and user information
- HTTPS Access
  - Multi-AZ with Traffic Manager

6.8. Data Traffic LoadBalancing

- If need a HTTPS listener, must deploy at least one SSL/TLS server certificate on SLB loadbalancer. The loadbalancer uses certificate to
  - Encrypt the traffic between application's users and loadbalancer (frontend connections)
  - Decrypt users' requests before sending them to backend servers.

6.9. Object Storage

- to store data without depending on a file system structure which makes it genuinely cross platform.
- As well as objects, related description of that object referred as metadata, are stored.
- This makes large data easy to handle across various systems.
- Using AWS S3 (Simple Storage Service), now a *de factor* standard in the market, data are widely accessible using simple HTTPS queries using Idendity and Access Management (IAM) for authentication. Can store data through a simple upload and retreive data by download.
- Objects are stored within containers referred as buckets.
- Specific features can be activated at bucket level (ex: versioning, replication, …)

Use Cases for Object Storage:

- unstructured data such as media and application files (archive or log files, pics)
- data sharing between users or computers (compatible with upload download data accesses)
- Based on S3 protocol, object storage is ideal for sharing data between apps.

6.10. Big Data Service

- a fast Hadoop provisioning in self-service mode.
- Through Big Data API, you can provision, manage and decommission an ephemeral Hadoop cluster in few hours instead of a couple of weeks

- Big Data is available to clients through a Restful API (direct call for automation purpose), cloud platform portal and CLI.

6.10.1. Tools

Big Data Service is based on HortonWorks HDP stack.

- HDFS
- MapReduce
- YARN
- Spark
- Hive
- HBase
- Ambari UI Console
- Knox Ranger
- Hue

6.10.2. Features

Through API, users can

- create big data HDP clusters
- manage clusters by adding/deleting workers or edge nodes
- retrieve inf about active clusters
- delete big data HDP clusters (they're ephemeral)
- create, delete, reset Unix users on a given cluster
- Install Hue on an edge node on given cluster

7. Hands On

7.1. Deploy stateless web application on Kubernetes

7.1.1. Step 1: Onboarding

- Create Kubernetes Namespace
  - isolated execution space for containers, on the worker nodes of Kubernetes cluster
- Create Routing Mesh/Ingress Connectivity
  - network connectivity, to expose Kubernetes services in your network zone
- Set up Kubernetes Environment

7.1.2. Step 2: Build PPSWEBAPP Docker image

- PPSWEBAPP is a demo RESTful API based on python flask framework and flask_restplus API module.
- Create Docker repository
- Build PPSWEBAPP Docker image

7.1.3. Step 3: Create TLS/SSL cerfiticates for PPSWEBAPP

- Create CSR config file
- Generate private key and CSR files
- Request TLS/SSL cerfiticate

7.1.4. Step 4: Deploy PPSWEBAPP dev instance on k8s cluster

- Deploy demo app TLS certificate
- Deploy 2 demo app replicas (2 pods)
- Deploy demo app service
- Test access to demo app service

7.1.5. Step 5: Expose non-prod demo app URL

- Deploy an ingress rule
- Test access to PPSWEBAPP non-prod URL from browser

7.1.6. Step 6: Promote PPSWEBAPP Docker image to PROD

- Run dev2prod service
- Check prod PPSWEBAPP repository

7.1.7. Step 7: Deploy PPSWEBAPP prod instance on k8s cluster

- Deploy demo app TLS certificate
- Deploy 2 demo app replicas (2 pods)
- Deploy demo app service
- Test access to demo app service
- Deploy an ingress rule
- Test access to PPSWEBAPP prod URL from browser

7.2. Terraform Deployment Goal

Azure services consumed by the applications are:

- Resource group: to consistently manage application resources
- VM: for computing resources
- Loadbalancer: to distribute incoming application traffic among compute instances
- PostgreSQL: relational database

Cloud Platform services consumed by applications are:

- Internet exposition through CIAP Hosting
- Internet access through CIAP Browsing
- Skynet alertign to detect abnormal activity or configuration
- Logs to ensure all activities are well reported
- OS Patching to get an overview of resources state of patching