UNIVERZITET SINGIDUNUM

Fakultet za informatiku i računarstvo

Cloud-Native rešenja

− diplomski rad −

Mentor:

prof. dr *Aleksandar Jevremović*

Kandidat:

Voja Đorđev

2015 / 201652

Beograd, *2020*.

Sadržaj

Uvod ..... **- 1 -**

1 Cloud-Native kao pojam ..... **- 2 -**

2 Infrastruktura u Cloud-u ..... **- 3 -**

2.1 Serverless ..... - 4 -

2.2 Pružaoci Cloud usluga (Cloud provajderi) ..... - 6 -

3 Kontejneri ..... **- 8 -**

3.1 Upravljanje kontejnerima (Container orchestration) ..... - 9 -

4 Cloud-Native aplikacije ..... **- 12 -**

4.1 Internet of Things - IoT ..... - 15 -

5 Baze podataka u Cloud-Native okruženju ..... **- 18 -**

6 Sloj otpornosti (Resilience layer) ..... **- 21 -**

7 DevOps koncept ..... **- 24 -**

8 CI/CD ..... **- 26 -**

9 Aspekt bezbednosti ..... **- 28 -**

9.1 The 4 C's of Cloud-Native Security ..... - 29 -

9.1.1 Cloud ..... - 29 -

9.1.2 Cluster ..... - 31 -

9.1.3 Container ..... - 31 -

9.1.4 Code ..... - 32 -

9.2 Detekcija – Logovi/Metrike i Monitoring ..... - 33 -

9.2.1 Logovi ..... - 33 -

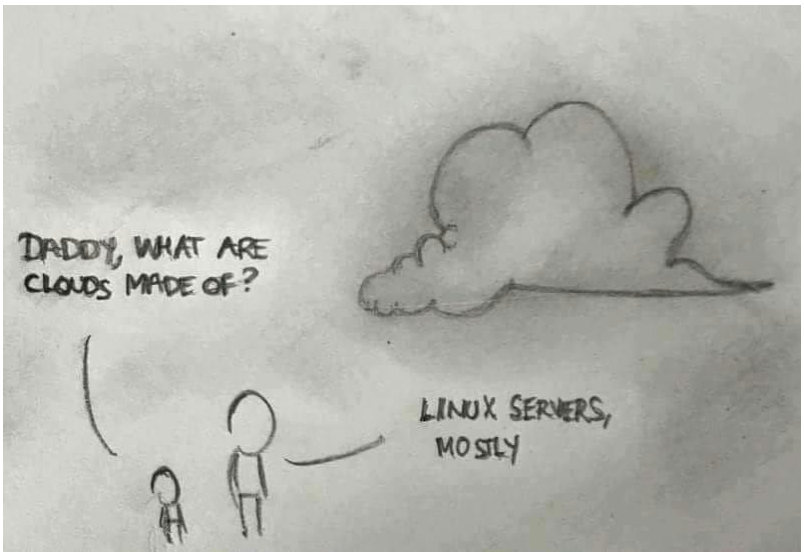
9.2.2 Metrike ..... - 35 -

10 Zaključna razmatranja ..... **- 36 -**

Literatura ..... **- 37 -**

Uvod

U poslednje vreme reč "*Cloud*" može se čuti na svakom koraku, ne samo među IT stručnjacima.



Slika 1 – Šaljiva definicija klauda (cloud)

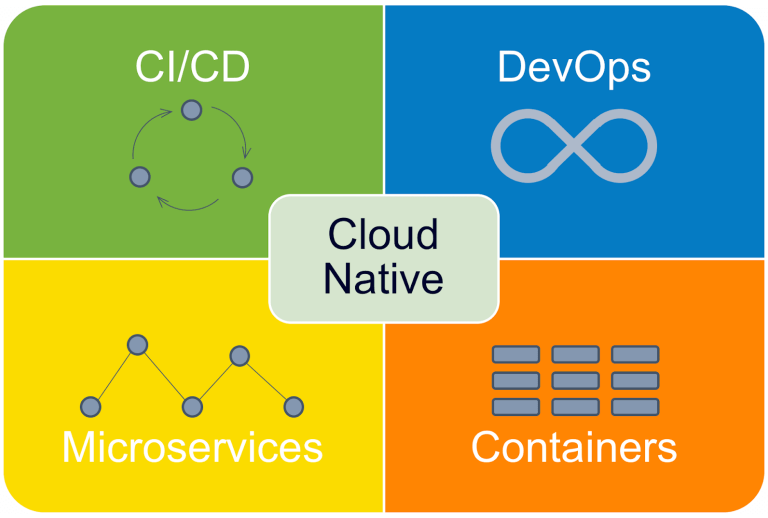
Ipak, u IT zajednici se koriste termini koji su malo određeniji. Jedan od takvih termina je "*Cloud-Native*".

Ovaj "trend" je veoma brzo dospeo i žižu IT zajednice. Postoje suprotstavljena mišljena oko toga da li će ovaj trend nestati istom brzinom kojom se pojavio ili će postati budućnost programiranja. Ono što je sigurno jeste da je Cloud-Native trenutno izuzetno polularan među IT stručnjacima te je stoga vredan izučavanja.

Cilj ovog rada jeste da detaljnije opiše ovaj pojam, odnosno trend, sa tehničke ali i organizacione perspektive, ukaže na prednosti i mane, imajući u fokusu aspekt bezbednosti.

Pored regularne literature, jedan od glavnih izvora informacija za ovaj rad je profesionalno iskustvo autora (9 godina u momentu pisanja ovog rada) stečeno radom u više kompanija, na različitim projektima, prisustvom različitim treninzima i konferencijama. Obzirom na prirodu teme i IT-a kao oblasti, objašnjenja i zaključci u ovom radu mogu biti podložni različitim interpretacijama. S tim u vezi, navode u ovom radu treba uzeti kao smernice i teme za razmišljnanje, a nikako kao striktna pravila

1 Cloud-Native kao pojam



Ilustracija 2 – Komponente koje čine Cloud-Native (jedan primer)

Postoji *više definicija* koje opisuju Cloud-Native:

- Cloud-Native je pristup razvoja i pokretanja aplikacija koje u potpunosti koriste sve prednosti *cloud computing* modela.
- Cloud-Native je pojam koji opisuje okruženja zasnovana na kontejnerima.
- Cloud-Native je način razvoja i pokretanja veoma skalabilnih aplikacija u modernim i veoma dinamičkim okruženjima kao što su javni, privatni ili hibridni cloud, oslanjajući se na najnovije (*cutting edge)* tehnologije.
- I slično...

Kao što se može primetiti iz priloženog, ne postoji jasna i jedinstvena definicija koja opisuje Cloud-Native. Ipak, može se primetiti da se određeni pojmovi ponavljaju: cloud computing, skalabilnost, dinamičnost, kontejneri, mikroservisi, DevOps, CI/CD, itd. Može se reći da su ovi pojmovi osnovni elementi koji sačinjavaju Cloud-Native pristup.

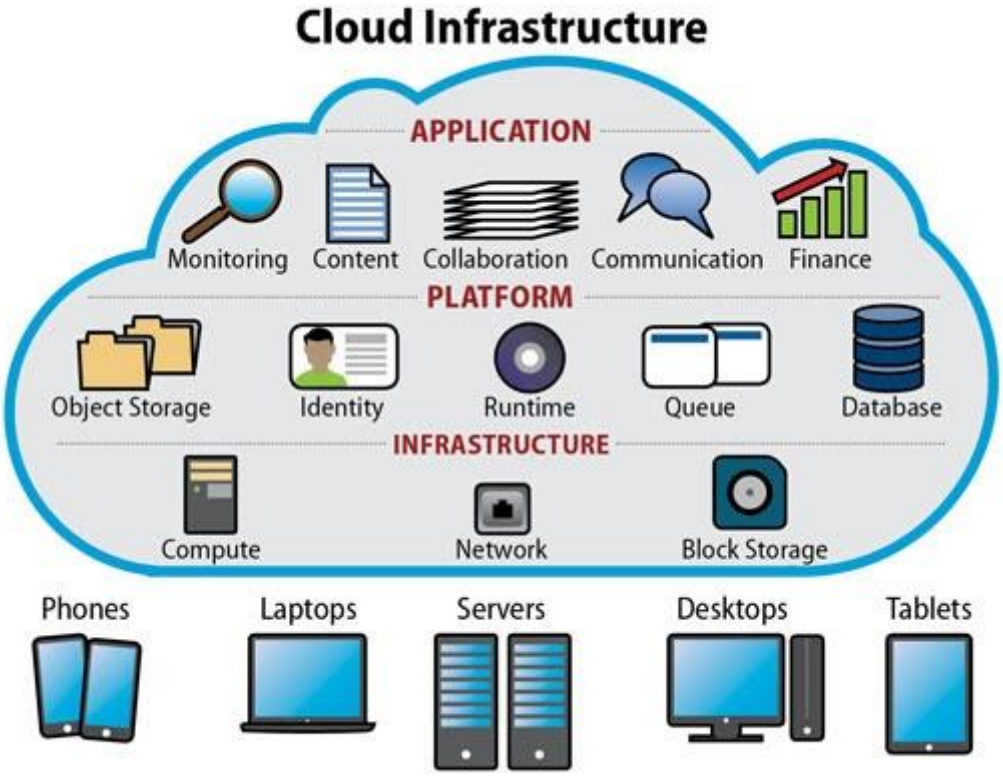
U nekom najopštijem posmatranju, Cloud-Native možemo opisati kao pristup u kome se bilo kakav razvoj (aplikacije, servisa, infrastrukture, itd.) planira i prilagođava za neko *apstraktno okruženje* (Cloud) gde je proizvod razvoja minimalno vezan za bilo kakav specifičan hardver, okruženje, tehnologiju ili pružaoca usluge.

Cloud-Native = Native for Cloud tj. <u>Napravljeno za Cloud</u>

Suštinski, Cloud-Native je jedan od pristupa koji, na svojstven način, pokušava da eliminiše čuvenu premisu “Radi na mom kompjuteru” (*It works on my computer*).

2 Infrastruktura u Cloud-u

Početni uslov za Cloud-Native pristup je postojanje apstraktnog, odnosno virtualnog, okruženja, poznatog kao *Cloud*.



Ilustracija 2 –Komponente koje čine infrastrukturu u Cloud-u

Kao što je ilustrovano iznad, IT infrastrukturu čine serveri, mrežni uređaji, skladišta podataka, itd. Specifičnost Cloud okruženja sastoji se u tome što pomenute fizičke uređaje zamenjuje virtualnim, i to posredstvom slojeva apstrakcije kao što su hipervizori, servisi, API, itd.

Postoji veliki broj prednosti ovakvog pristupa infrastrukturi:

- Eliminiše se rizik od fizičkih kvarova opreme
- Hardver je u startu skup dok se većina cloud komponenti plaća *per use* (prema realnom korišćenju)
- Hardver je potrebno planirati unapred dok se cloud instance infrastrukture mogu dobiti u nekoliko klikova mišem
- Cloud resursi se mogu maksimalno iskoristiti
- itd.

Iz svega navedenog, jasno je zašto je cloud okruženje dobar izbor.

Naravno, postoje i određeni nedostaci:

- Takozvani "*Vendor lock-in*" odnosno nemogućnost jednostavne promene vendora, odnosno pružaoca cloud usluga, jer su sve aplikacije i servisi koje koriste suviše usko vezani za specifičnog vendora
- Čuvanje osetljivih podataka na Cloud-u nije prihvatljivo
- Isti servisi su implementirani na drugačiji način kod različitih vendora
- itd.

Ipak, evidentno je da se IT kompanije sve više okreću korišćenju infrastrukture u Cloud-u, bez obzira na pomenute nedostatke.

Na osnovu deployment modela, Cloud se može klasifikovati kao:

- *Javni*
- *Privatni*
- *Hibridni*
- *Community*

Na osnovu vrste servisa koji Cloud nudi, razlikuju se:

- *IaaS* - Infrastructure as a Service
- *PaaS* - Platform as a Service
- *SaaS* - Software as a Service
- ili Storage, Database, Information, Process, Application, Integration, Security, Management, Testing-as-a-service

Obzirom da je Cloud izuzetno široka tema, nećemo zalaziti dublje u funkcionisanje svih pojedinačnih oblika Cloud-a.

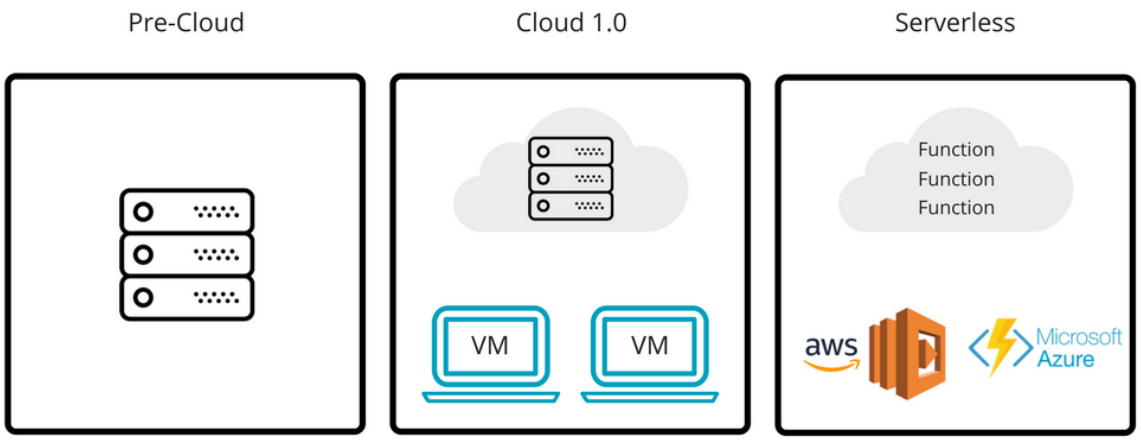
2.1 Serverless

Veoma interesantan oblik Cloud okruženja, koji je definitivno vredan pomene u smislu Cloud-Native pristupa, jeste *Serverless*.

U ovom obliku Cloud okruženja, infrastruktura je maksimalno apstrahovana tako što provajder upravlja resursima i dinamički ih dodeljuje aplikacijama kada je to potrebno. Kompletna infrastruktura zasnovana je na *funkcijama*.

Naplata se vrši samo na osnovu resursa koje je aplikacija koristila prilikom izvršavanja.

Može se reći da je Serverless novija generacija Cloud-a, kao što je ilustrovano na slici ispod.



Ilustracija 3 –Evolucija Cloud-a

Neke od prednosti ovog pristupa, specifične u odnosu na Cloud, su:

- Uvek je dostupno tačno onoliko resursa koliko je potrebno. Nije neophodno planirati resurse unapred. Npr. Start-Up koji ne želi da troši mnogo novca za infrastrukturu u početnoj fazi (proizvod još nije završen i još nije popularan) platiće samo malu cifru za mali broj izvršavanja koda svoje aplikacije. Isto tako, kada proizvod odmakne u razvoju i postane popularan (više korisnika ga koristi) resursi će automatski, i bez posebnog skaliranja, biti dostupni da podrže veći broj korisnika. Ovo će, naravno, podrazumevati veće troškove izvršavanja ali pretpostavka je da u ovom trenutku veće korišćenje aplikacije podrazumeva i veći prihod od iste koji bi, uz dobar poslovni model, trebalo da bude veći od svih troškova, uključujući i troškove izvršavanja koda.
- Planiranje troškova je mnogo efikasnije. Obzirom da se naplata vrši na osnovu resursa koji su potrebni da bi se kod izvršio, može se vrlo precizno odrediti koliko neki "*bug*" košta (ne-efikasan kod će uzeti više resursa da bi kompenzovao). Ovo daje potpuno novi pogled na održavanje koda. Može se, na primer, zaključiti da određeni *bug* jako malo umanjuje troškove dok bi, sa druge strane, cena otklanjanja istog prilično velika. U ovom slučaju se može doneti odluka da se *bug* prosto ne otklanja, naravno pod uslovom da isti ne utiče na "*User Experience*", tj. nije vidljiv i očigledan krajnjim korisnicima.
- Mogućnost skaliranja resursa je mnogo jednostavnija i ne zahteva intervenciju system administratora

2.2 Pružaoci Cloud usluga (Cloud provajderi)

Postoji veliki broj kompanija koje se bave profesionalnim održavanjem i *iznajmljivanjem* infrastrukture. Među tim uslugama su iznajmljivanje fizičkih servera, kolokacija (iznajmljivanje prostora za postavljanje opreme) i Cloud usluge.

Među svim vendorima (provajderima) Cloud usluga, najviše se ističu:



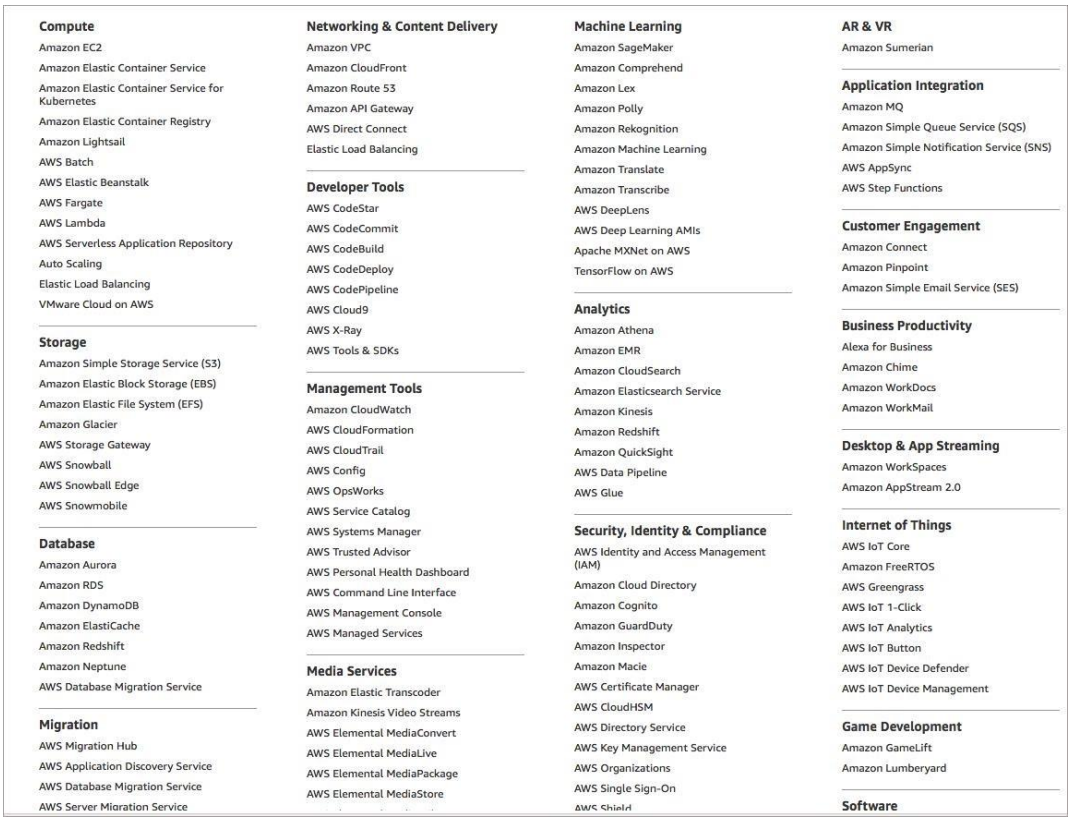
Ilustracija 4 – Najistaknutiji Cloud provajderi

Svakako postoje i drugi Cloud provajderi (pružaoci Cloud usluga), kao što su:

- *PhoenixNap,*
- *Rackspace,*
- *Digital Ocean,*
- *IBM Cloud,*
- *Alibaba Cloud*
- i slično.

Manje provajdere odlikuje uži spektar različitih proizvoda i servisa ali su zato često jeftiniji i usmereniji na klijente tako da se odvih provajdera često može dobiti kvalitetnijua usluga, specifična za svakog klijenta.

Svi provajderi imaju veliki broj proizvoda i usluga koji na različite načine ispunjavaju slične potrebe klijenata u smislu Cloud infrastrukture. Malo je reći da postoji veliki broj servisa koji se koriste. Na idućoj strani, ilustrovan je pregled AWS servisa iz 2018. godine.



Ilustracija 5 – Pregled AWS servisa iz 2018. godine

Svi provajderi teže globalnoj dostupnosti svih svojih servisa ali ipak postoje razlike zavisno od regiona u kom se nalazi fizička oprema provajdera ali i lokacije korisnika usluge. Česta prepreka su i zakonske regulative.

Pored servisa i proizvoda, Cloud provajderi nude i *API pristup*. Konkretno, na primeru AWS-a, može se koristiti *aws-cli* za interakciju sa Cloud provajderom preko komandne linije. Ovakav pristup omogućava upravljanje servisima i proizvodima na AWS-u putem koda koji koristi aws-cli. Ovaj koncept nosi naziv *IaaC (Infrastructure as a Code)*. Ovo znači da, umesto tradicionalnog podešavanja infrastrukture, sistem administratori pišu kod koji upravlja infrastrukturom. Npr. umesto da se ručno podiže ili konfiguriše identičan server za Dev, Test i Produkcijsko okruženje, jednom se napiše kod koji kreira server, pritom prima različite parametre koji se razlikuju na osnovu okruženja, i 3 puta pokreće kako bi se server automatski kreirao.

Prednosti ovakvog pristupa su brojne:

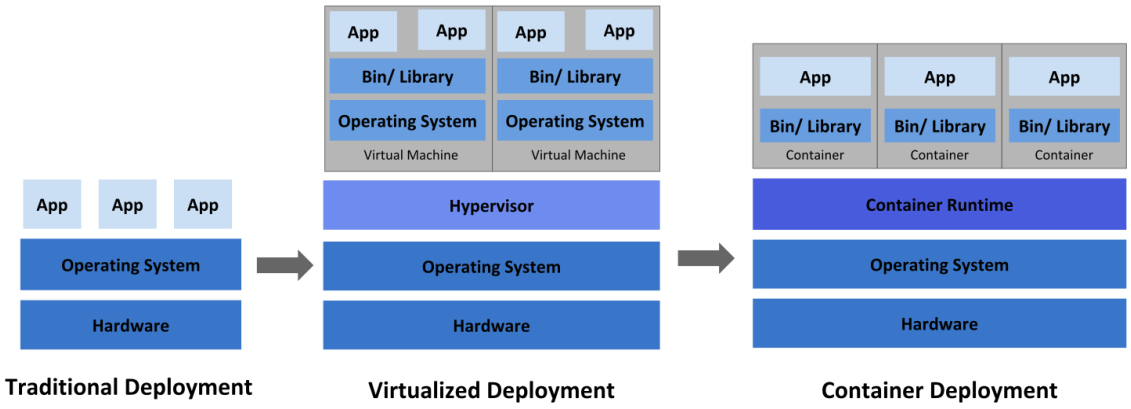
- Automatizacija i eliminisanje ljudske greške
- Olakšava se upravljanje velikom infrastrukturom - promene se vrše samo kroz kod koji se nalazi u nekom sistemu za upravljanje verzionisanjem (najčešće GIT) tako da se akcije administratora bolje usklađuju.
- Putem dodatnih servisa (npr. *ChatOps*) može se developerima dati mogućnost da sami upravljaju infrastrukturom koja im je potrebna, i to na kontrolisan i bezbedan način
- Itd.

Neki od alata koji se koriste za IaaC su *Ansible*, *Chef*, *Puppet*, i slično.

3 Kontejneri

U osnovi Cloud-Native tehnologija su kontejneri.

Kontejneri daju mogućnost da se, kroz specifičnu *hosted* virtualizaciju, pokreću aplikacije i njihovi *depencency*-ji (dodatne komponente od kojih zavisi aplikacija) u izolovanom okruženju, na bezbedan način i bez uticaja na druge kontejnere i procese unutar istog operativnog sistema (OS).



Ilustracija 6 – Poređenje tradicionalnih, VM i kontejner okruženja

Pre kontejnera, u ovu svrhu su se koristile virtualne mašine (VM). Dobar primer klasičnog hosted hipervizora (sloj virtualizacije) koji omogućava podizanje virtualnih mašina na OS-u je *Oracle Virtual Box*.

Funkcionalno gledano, kontejneri se ponašaju veoma slično kao VM ali, umesto kompletne VM sa celim operativnim sistemom unutar nje, kontejner se pokreće kao *Linux proces* unutar kernela. Ovo znači da kontejneri sadrže samo kod i depedency-je koji su potrebni za pokretanje aplikacije i to ih, u odnosu na VM, čini mnogo manjim i lakšim za transport i skaliranje a odlikuje ih i mnogo brže pokretanje (start up).

Kontejner je definisan kao "Linux proces" sa razlogom. Izvorno, kontejneri su *feature* (*funkcionalnost*) Linux kernela. Konkretno, sve kontejner tehnologije se zasnivaju na *CGrupama* (*Control Groups*) o kojima se više može pročitati na: https://en.wikipedia.org/wiki/Cgroups [posećeno dana: 26.12.2019.]

Iako je poslednjih godina bilo dosta pomaka u Windows kotejnerima, za pokretanje kontejnera u produkcijskim okruženjima je i dalje većinom neophodno bude uključen Linux kernel, bar na neki način.

Ubedljivo najpoznatiji kontejneri, iako definitivno ne jedini, su *Docker* kontejneri. Container runtime koji koristi docker je *containerd*. Docker, ali i druge kontejner tehnologije, imaju 2 glavne komponente:

- *Image* - fajl koji se sastoji iz više slojeva i predstavlja polaznu tačku koja definiše okruženje kontejnera (šablon)
- *Container* - izvršna instanca, pokrenuta na osnovu nekog image-a i dodatnih runtime (izvršnih) parametara.

Iz perspektive developera, Image i Container se mogu uporediti sa odnosom Klase i Objekta, gde je Klasa recept ili šablon and objekat instanca klase sa konkretnim vrednostima atributa.

Da se nadovežemo na prethodni pasus o neophodnosti Linux kernela objasnićemo kako funkcioniše *Docker for Windows*. Prilikom instalacije ovog sistema na Windows-u neophodan korak je uključivanje VMWare virtualizacije. Ovo je neophodno jer "Docker for Windows" koristi VMWare hipervizor da bi podigao VM sa Linux OS-om i postavio Docker server na njega. Docker klijent, koji se instalira na Windows OS-u, zatim komunicira sa Docker server-om preko virtualne mreže, konkretno preko virtual ethernet interfejsa.

Za verzije Windowsa koje nemaju VMWare postoji mogućnost instalacije pomoću alata pod nazivom *Docker Toolbox*. Princip je veoma sličan i glavna razlika je što se, umesto VMWare-a, kao hipervizor koristi Oracle Virtual Box na kom se onda podigne Linux VM sa Docker server-om.

Bitno je pomenuti da, pored kontejnera (kao i image-a, neophodnih za kontejnere), Docker sistem radi i sa drugim objektima:

- *Docker network* - virtualna mreža koja može dodatno izolovati kontejnere unutar istog Docker sistema i olakšati međusobnu komunikaciju kontejnera
- *Docker volumes* - virtualni storage kojim upravlja Docker sistem i koji se može koristiti od strane kontejnera radi postizanja perzistencije podatka

Neki od drugih kontejnera su CoreOS rkt, Mesos Containerizer, itd.

3.1 Upravljanje kontejnerima (Container orchestration)

Da bi se kontejneri iskoristili na pravi način, potrebno je definisati i podići (startovati) veliki broj kontejnera, definisati njihove međuzavisnosti, skalirati ih, itd. Zapravo, potrebno je upravljati kontejnerima "na veliko". Ovo se postiže korišćenjem alata koji su specijalizovani za upravljanje kontejnerima.

Jednostavnosti radi, ovi alati biće predstavljeni na primeru Docker-a.

Sam Docker ima sopstvene alate za upravljanje kompleksnim sistemima kontejnera:

1. *Docker Compose*

   Ovaj alat kao šablon koristi YAML fajl u kom se definišu servisi, pri čemu se svaki servis može sastojati iz jednog ili više kontejnera, i njihove međuzavisnosti.

2. *Docker Swarm*

   Ovaj alat predstavlja nadogradnju na Docker compose tako što omogućava distribuiran Docker sistem u obliku cluster-a (raspoređen na više servera). I dalje se koriste YAML fajlovi Docker Compose-a ali se sada

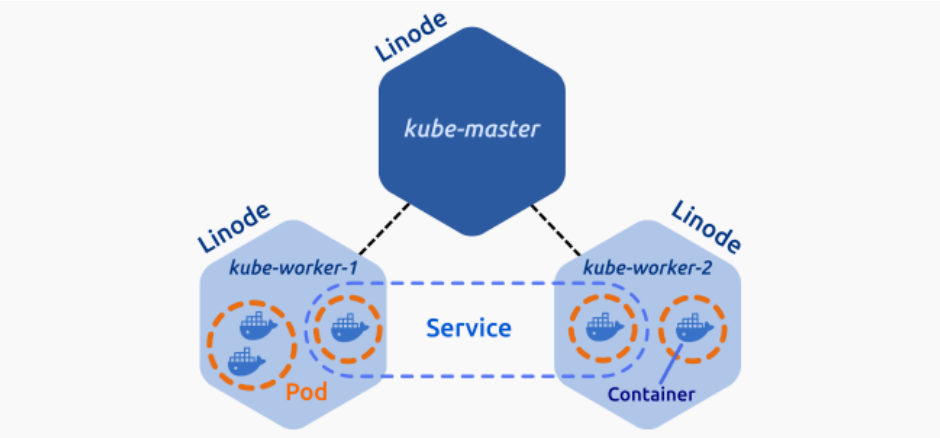
kontejneri mogu podizati na različitim serverima pri čemu se postiže veća dinamičnost sistema i manja zavisnost od potpornog OS-a, odnosno dobija se HA (High Available) sistem. Što se samih kontejnera i servisa tiče, dobija se mogućnost *horizontalnog* skaliranja, odnosno proširivanje opsega dostupnih resursa dodavanjem nobih servera umesto povećavanja resursa postojećih (vertikalno skaliranje). Postoji niz dodatnih funkcija koje omogućavaju kreiranje i upravljanje veoma kompleksnim sistemom na jednostavan način.

Postoje i drugi sistemi, koji su specijalizovani za upravljanje kontejnerima, među kojima su i Docker kontejneri. Svakako najpopularniji i najzastupljeniji je *Kubernetes*.

Kubernetes open-source je platforma razvijena od strane Google-a (projekat je započet 2014. godine) koja služi za automatizaciju operacija nad kontejnerima, kao što su razvoj, raspoređivanje i skalabilnost u klasteru.

Bez ulaženja u previše detalja, komponente Kubernetes klastera, u smislu arhitekture, su:

- *Master node* (jedan ili više) – server koji sadrži komponente za upravljanje sistemom
- *Worker node* (najčešće 2 ili više) - server koji sadrži komponente za pokretanje kontejnera



Ilustracija 7 – Osnovne komponente Kubernetes sistema – arhitektura klastera i organizaciaj kontejnera

U smislu organizacije kontejnera, takođe bez ulaženja u previše detaljna, osnovne komponente su:

- *Pod-ovi* - logička jedinica koja se može sastojati od jednog ili više kontejnera
- *Servisi* - definišu veze pod-ova sa drugim objektima unutar i izvan klastera

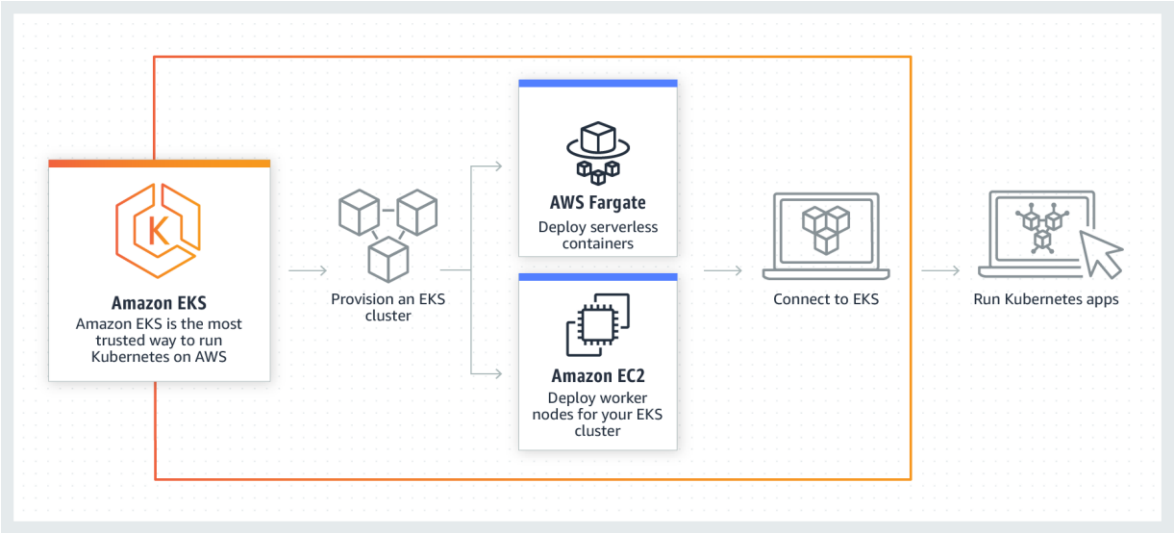
Postoji još mnogo detalja u smislu servisa i komponenti koje se koriste na Master i Worker nodovima, kao i različitih objekata koji se koriste unutar sistema. Obzirom da je Kubernetes veoma široka tema, koja pritom nije glavni cilj ovog rada, nećemo zalaziti dublje u funkcionisanje samog Kubernetes-a.

Bitno je napomenuti da je glavna prednost Kubernetes-a to što je, kao open-source platforma sa velikom primenom i podrškom u IT zajednici, postao *standard za upravljanje kontejnerima*.

Funkcionalno gledano, Kubernetes se ne razlikuje mnogo od Docker Swarm-a ali je zato univerzalniji i može raditi i sa drugim kontejner runtime-ovima osim containerd. Takođe, obzirom na podršku zajednice, ima mnogo više funkcionalnosti nego Docke Swarm.

Razni Cloud provajderi su inkorporirali Kubernetes u svoje sisteme do te mere da nude Kubernetes kao servis (*AWS EKS*, *AKS*, etc.). Prednosti su brojne ali ističe se svakako jednostavnost kreiranja klastera, upravljanje i održavanje master i worker nodova, kao i upravljanje drugim servisima unutar datog Cloud vendora, koji su neophodni dodatak na servise unutar Kubernetes sistema, na automatizovan način iz samog Kubernetes sistema. Na primer:

> AWS EKS u startu zahteva kreiranje role koja dozvoljava EKS-u da upravlja drugim resursima unutar datog AWS naloga. Ovo je neophodno da bi se npr. kreiranjem servisa tipa Load Balancer unutar Kubernetes sistema, radi expose-ovanja neke aplikacije spoljašnjem svetu, automatski kreirao neophodni Load Balancer na AWS-u, uz odgovarajuće target grupe ka svim worker nodovima, i da bi se ažurirale security grupe ka svim worker nodovima kako bi propustile konekciju do aplikacije.
>
> Na slici ispod ilustrovan je jednostavan process pokretanja i korišćenja Kubernetes-a kao servisa na AWS-u.



Ilustracija 8 – Dijagram pokretanja i stavljanja u upotrebu AWS EKS (Elastic Kubernetes Service)

4 Cloud-Native aplikacije

U suštini, tradicionalne aplikacije su optimizovane za polje koje odlikuje *predvidljivost*. Tradicionalni pristup se fokusira na sve što je poznato, istorijski, pritom prilagođavajući se za digitalni prostor.

Sa druge strane, Cloud-Native aplikacije se fokusiraju na prilagođavanje promenljjivim zahtevima digitalnog sveta, eksperimentisanje i rešavanje novih problema.

Neke od osnovnih razlika u karakteristikama oba pristupa su:

1. *Predvidljivost (Predictability)*

   Tradicionalnim aplikacijama je potrebno mnogo vremena da se build-uju (kompajliraju), posebno kad se uporede sa istim aplikacijama napisanim kao Cloud-Native. Obično se release-uju (objavljuju) kao jedan veliki paket i nema mnogo mesta za bilo kakve izmene ili skaliranje. Ovo znači da, kada se u realnom (produkcijskom) okruženju pojavi problem koji nije bio predviđen prilikom razvoja i testiranja, je rešavanje istog jako sporo i rizično.

   Cloud-Native odgovara framework-u koji je dizajniran za maksimalnu otpornost. Obzirom da se sastoji od više manjih komponenti (kontejnera), koje se ponašaju uvek na isti način i moguće je njima upravljati na bezbedan i koordinisan način, relativno je lako uraditi izmene na produkciji kada se pojave nepredviđeni problemi

2. *Zavisnost od OS-a (OS dependency)*

   Arhitektura tradicionalnih aplikacija dozvoljava direktnu zavisnost između aplikacija i OS-a. Upravo iz ovog razloga migracija, ili skaliranje, tradicionalnih aplikacija predstavlja veoma kompleksan i rizičan proces.

   Arhitektura Cloud-native aplikacija dozvoljava developerima da koriste platforme i kontejnere radi apstrahovanja OS dependency-a (aplikacija se pokreće unutar kontejnera koji je uvek isti, bez obzira gde se pokreće). Primarni cilj je da se fokus usmeri na sam softver a smanji briga o okruženju.

3. *Interaktivnost*

   Tradicionalne aplikacije rade u “Silosima”. Kada je aplikacija gotova, glavni prioritet je pokretanje iste na produkciji, pri čemu se zapostavlja vrednost i kvalitet aplikacije. Nema mnogo mesta za ispravke nakon što je aplikacija gotova. Može se reći da se razvoj tradicionalnih aplikacija odvija po tzv. “Waterflow” pristupu.

   Cloud-Native aplikacije su otvorenije za saradnju. DevOps princip, skup ljudi, procesa i alata, ovde omogućava mnogo brži i jednostavniji prenos završenog koda na produkciju. Izmene je moguće vršiti mnogo češće, čim se uoče problemi. Može se reći da se razvoj Cloud-Native

aplikacija odvija po Agile development pristupu - u više iteracija, tj. čest deployment gotove aplikacije.

4. *Nivo automatizacije*

Tradicionalne aplikacije odlikuje nizak stepen automatizacije. Veliki broj akcija se obavlja ručno, od strane operatora, što povećava mogućnost ljudske greške. Razni pokušaji automatizacije su veoma skupi, spori ali i neefikasni jer arhitektura samog sistema ne uzima u obzir promenljivost okruženja i spoljnih faktora.

Cloud-Native aplikacije odlikuje visoka dinamičnost i skalabilnost. Sama arhitektura je dizajnirana tako da umesto jedne velike i robusne aplikacije postoji više malih komponenti kojima se koordinira na jednistven način koji sam po sebi podrazumeva unapred deifinisane načine za izmene pojedinačnih komponenti i skaliranje celog sistema ili samo njegovih delova.

Radi boljeg objašnjenja prelaska sa tradicionalnog na cloud-native pristup, uzećemo za primer tradicionalni razvoj i pokretanje java aplikacije, pokušaj startovanja iste te aplikacije u okviru kontejnera i na kraju razvoj iste takve aplikacije u *Quarkus* framework-u, od starta predviđenu za rad u Docker kontejnerima.

*U svakom slučaju, sam kod se piše na isti način.

U tradicionalnom pristupu, gotova aplikacija se pokreće "java -jar <app>.jar" komandom. Potrebno je da OS na kom se pokreće ima JRE (Java Runtime Environment), i sve dependency-e koji su potrebni za aplikaciju. Ovo se obično postiže korišćenjem Maven-a ili nekog drugog package managera. Aplikacija se tada pokreće u okviru Java Virtualne Mašine. Ovo je zapravo, po strukturi, veoma sličan pristup kao Cloud-Native. Može se pokrenuti i više aplikacija, svaka u svojoj Java Virtualnoj Mašini.

Problem nastaje kada jedna od aplikacija na istom OS-u zahteva dependency koji je u sukobu sa ostalim aplikacijama ili njihovim dependency-jima. Veoma prost primer je da razvijate aplikaciju koja traži Java 9 dok ostale aplikacije rade na Java 8 i ne mogu se pokrenuti na Java 9.

Takođe morate voditi računa o resursima. Vrlo pažljivo morate konfigurisati JRE kao i same aplikacije da bi mogle sve zajedno da funkcionišu na istom sistemu.

Migracija je dosta rizična i zahtevna jer morate podesiti sistem (instalirati Javu i dependency-je) identično, što najčešće mora da se uradi ručno, od strane system administratora.

Sve ove probleme mogu rešiti kontejneri.

Svakako, moguće je jednostavno java aplikaciju izbildovati i zatim prosto .jar fajl ubaciti u kontejner koji je napravljen tako da podržava java okruženje. U ovom slučaju se

aplikacija pokreće unutar JVM-a (Java Virtualne Mašine), koji se nalazi unutar Docker kontejnera.

Ovakav pristup je u nekoj meri prihvatljiv i predstavlja korak ka Cloud-Native okruženju, mada ne sasvim. Obzirom da aplikacija nije od početka pisana za pokretanje unutar kontejnera, ne mogu se u potpunosti iskoristiti sve mogućnosti. Postoje i određene komplikacije, kao na primer:

> Java 8 generalno ne funkcioniše dobro sa Docker-om. Problem proizilazi iz činjenice da Java, kao i JVM, postoje duže nego Docker kontejneri, containerd i CGrupe. JVM unutar Docker kontejnera nije svestan da se nalazi u kontejneru već vidi kompletne resurse underlying (potpornog) OS-a. Tako da, ako nisu postavljeni specifični parametri koji ograničavaju resurse koje JVM uzima, JVM će po podrazumevanim (defaultnim) podešavanjima pokušati da uzme maksimalne resurse, dostupe na OS-u.
>
> Ovo znači da, u slučaju da se na istom serveru pokrenu 2 java aplikacije, svaka unutar posebnog Docker kontejnera, obe će, u slučaju visokog load-a, pokušati da uzmu maksimalne resurse OS-a. Pritom, nesvesne jedna druge, obe aplikacije će u zbiru pokušati da uzmu duplo više resursa nego što je dostupno bez da se *Garbage Collector* uopšte aktivira. U nekom momentu, OS će ostati bez resursa i prekinuće jednu od ove 2 aplikacije nasilno posredstvom OOM (Out Of Memory) Killer mehanizma.
>
> Slična stvar će se desiti ako se postavi limit na Docker kontejnerima. JVM će i dalje pokušati da uzme maksimalne resurse ali će u ovom slučaju umesto na OS limit, naići na Docker kontejner limit.
>
> Jedini workaround () jeste da se precizno definišu JVM limit-i u smislu HEAP memorije (parametri Xmx i Xms) ali i drugi parametri. Ovo nije veoma praktično rešenje i teško ga je sprovesti kada se radi o dinamičkom i distribuiranom sistemu sa velikim brojem kontejnera. Razlog je što je uglavnom nemoguće predvideti tačne resurse koji su neophodni i koji su dostupni u određenom momentu.
>
> Ovaj problem je rešen u novijim verzijama Jave kao i u Java 8, i to sa update-om 191, dostupnim na sledećoj stranici:
>
> https://www.oracle.com/technetwork/java/javase/8u191-relnotes-5032181.html [posećeno dana: 26.12.2019.]
>
> Ovaj update je izašao u Aprilu 2019. godine. Ipak, ovaj primer prilično dobro ilustruje probleme koji nastaju kada se aplikacije od početka ne piše za Cloud-Native.

U slučaju Cloud-Native pristupa, kada se koristi *Quarkus* framework, zaobilazi se JVM.

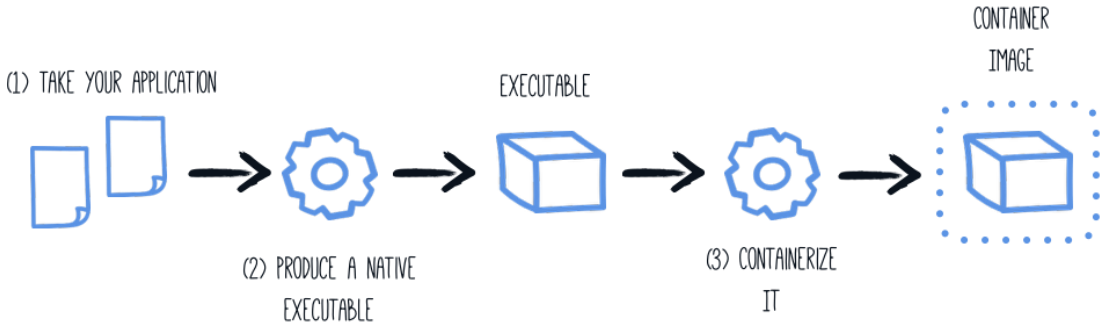
Quarkus koristi tzv. *GraalVM* za kompajliranje java koda po principu *ahead-of-time (AOT)* u native executable. Ovo znači da se kod kompajlira direktno u mašinski kod.

Dalje, ovo znači da se aplikacija, umesto u *Java HotSpot VM*, pokreće u drugačijoj implementaciji VM poznatoj kao *Substrate VM*.

Ovo menja tradicionalni pristup jave u potpunosti jer je izvorna ideja JVM-a bila pokretanje java koda u bilo kom okruženju. Ipak, kontejneri mogu zameniti JVM pristup u potpunosti a pritom otvaraju vrata ka velikom broju novih mogućnosti. U modernom Cloud-Native pristupu, kao krajnji proizvod se dostavlja kontejner a ne sama aplikacija, tako da JVM unutar kontejnera postaje suvišan.

Krajnji proizvod Quarkus framework-a je kontejner koji zautima mnogo manje prostora i pokreće aplikaciju koja ima znatno kraće start-up vreme i zahteva mnogo manje resursa, a pritom se mogu iskoristiti sve prednosti rada sa kontejnerima kao i alati za upravljanje (orkestraciju) kontejnerima, kao što je Kubernetes.

Na slici ispod, prikazani su osnovni koraci prilikom build-ovanja java aplikacije posredstvom Quarkus framework-a.



Ilustracija 9 – Proces build-ovanja aplikacije posredstvom Quarkus framework-a

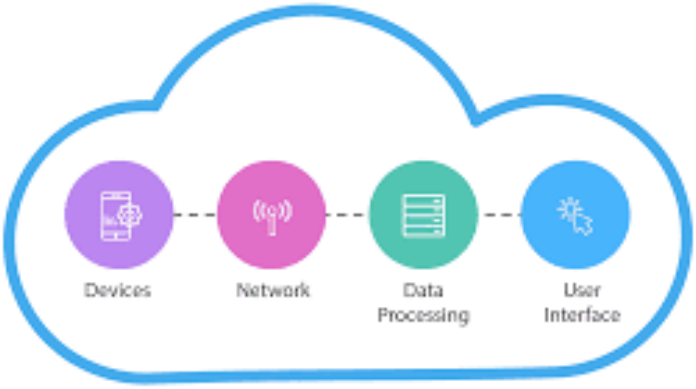
4.1 Internet of Things - IoT

Trend *Internet Of Things (IoT)* je poslednjih godina sve popularniji. Naime, radi se o trendu povezivanja "stvari" na internet radi prikupljanja podataka, obrade, udaljene kontrole i sl.



Slika 2 – Prikaz IoT sistema

Kompletan IoT system se sastoji iz 4 osnovne komponente, koje su ilustrovane na slici ispod.



Ilustracija 10 – Osnovne komponente IoT sistema

1. *Senzori (uređaji)*

   Senzori prikupljaju podatke iz okruženja. Više senzora može biti povezano zajedno ili senzori mogu biti samo deo nekog uređaja koji ima i druge funkcije osim prikupljanja podataka. Primer za ovo bi bile kamere za nadzor ili senzori temperature i vlažnosti u stambenim ili poslovnim prostorijama.

2. *Konektivnost*

   Prikupljeni podaci se šalju u Cloud putem interneta. Uređaji se mogu povezati sa internetom na više načina u koje spadaju: SIM kartice, satelit, WiFi, Bluetooth, LPWAN, ethernet, itd. Kućni uređaji se obično povezuju preko kućne WiFi ili Ethernet konekcije dok se pokretni uređaji najčešće povezuju preko SIM kartice ili satelita (GPS tracking), često uz upotrebu VPN tunela.

   Za povezivanje se najčešće koristi *MQTT* protokol, posebno dizajniran za M2M (Machine to Machine) tj. IoT komunikacije. Ovaj protokol je koncipiran keo veoma lagan (poruke male veličine) transportni protokol koji radi po sistemu *publish/subscribe (objavi/pretplati se)*. Više informacija o samom MQTT-u, dostupno je na zvaničnom sajtu: http://mqtt.org/ [posećeno dana 09.01.2020.]

3. *Obrada podataka*

   Kad su podaci prikupljeni, potrebno ih je obraditi, doneti neke zaključke i preduzeti određene akcije, zavisno od infromacija koje su formirane obradom. Ovo mogu biti jednostavne informacije kao što je saznanje koja je trenutna temperatura u stanu ali mogu biti i kompleksnije akcije kao što korišćenje kamere za identifikaciju objekata, npr. provalnika u stanu.

4. *Korisnički interfejs*

Na kraju, informacija treba biti predstavljena korisniku na neki način da bi on mogao da je iskoristi. Ovo može biti npr. notifikacija kao što je email, SMS, poziv i sl. Recimo, sistem može poslati SMS IT timu kada je temperatura u server sali suviše visoka.

Korisnik takođe može imati mogućnost da proaktivno proverava sistem. Recimo kada korisnik hoće da pogleda live (uživo) video prikaz sopstvenog stana, to može uraditi preko aplikacije na telefonu ili internet pretraživača (browsera) na kompjuteru.

Specifičan slučaj je kada korisnik želi da preduzima neke akcije na sistemu (udaljeni pristup). Na primer, klima uređaj koji se priključen na kućni WiFi daje korisniku mogućnost da mu pristupi udaljeno, preko aplikacije na telefonu, i uključi hlađenje/grejanje par sati pre nego što će osoba stići kući.

U kombinaciji sa prikupljanjem velike količine podataka i primenom statistike nad njima (BigData, Machine Learning, i slično) mogu se naći primeri veoma inovativne primene IoT tehnologija. U nastavku je navedeno nekoliko takvih primera:

- Računanje polise osiguranja vozila na osnovu stila vožnje korisnika, do koga se dolazi prikupljanjem i obradom GPS podataka koje emiture uređaj iz automobila: https://www.msg-global.com/iota [posećeno dana: 09.01.2020.]

- Optimizacija upravljanja otpadom kroz prikupljanje informacija od strane pametnih kanti za smeće i kontejnera, na solarno napajanje: https://bigbelly.com/platform/ [posećeno dana: 09.01.2020.]

- Pravilnije planiranje treninga i poboljšanje fizičke spremnosti (fitness) na osnovu podataka (puls, kretanje, itd.) prikupljenih od strane pametnih satova i narukvica (wareables): https://www.fitbit.com/eu/home [posećeno dana: 09.01.2020.]

Ono što je posebno interesantno je činjenica da je IoT moderan i relativno nov pristup u vrlo sličnoj meri kao Cloud-Native. Dakle, IoT nikada nije imao tradicionalni pristup primene već se razvijao direktno za Cloud, možda čak i praleleno sa Cloud-Native pristupom uopšte. Dakle, može se reći da je na primeru IoT tehnologija maksimalno iskorišćen Cloud-Native pristup u pravom smislu reči.

5 Baze podataka u Cloud-Native okruženju

Prednosti dizajniranja aplikacija za Cloud-Native su prilično jasne ali to možda nije tako sa bazama podataka.

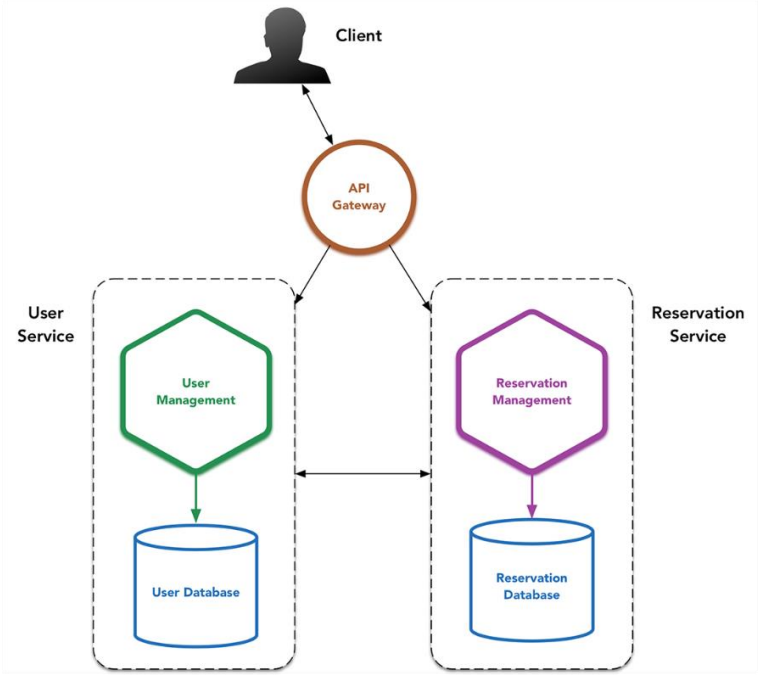
Kocept kontejnera daje mnogo mogućnosti ali takođe podrazumeva kratkoročnost ili nestalnost na nivou pojedinačnog kontejnera. Naime, aplikacija je stabilna dokle god je pokrenut određen minimalni broj kontejnera koji je neophodan za obavljanje svih akcija ali pojedinačni kontejneri mogu se neplanirano ugasiti (fail-ovati - ovo je očekivana pojava), pri čemu će u nekom momentu biti zamenjeni novim kontejnerima na osnovu pravila skaliranja koja su zadata u sistemu.

U svakom sistemu ili aplikaciji, podaci su najvažniji faktor. Tradicionalne baze podataka, koje odlikuje visoka stabilnost, ne uklapaju se idealno u Cloud-Native koncept.

Osnovni problem sa tradicionalnim bazama je to što, zbog važnosti podataka, postaju prevelike.

Tradicionalne, monolitne, aplikacije i sistemi obično svi rade sa jednom velikom bazom podataka. Do ovakve situacije dolazi tako što se proširivanjem sistema, iz praktičnih razloga, prosto dodaju nove tabele u postojeću bazu podataka kao proširenje već postojećeg modela podataka. Taj pristup je veoma logičan ali, dugoročno gledano, na kraju dovodi do formiranja jako velike baze podataka koja u nekom momentu postaje kritična masa koja je vrlo nezahvalna za održavanje i eksponencijalno gubi na performansama. U trenutku kada nastane ovakva situacija veoma je teško promeniti pristup i rešiti problem.

Cloud-Native pristup podrazumeva, umesto jedne velike baze podataka, formiranje više malih baza koje odgovaraju pojedinačnim servisima koji ih koriste. Ovo se može nazvati "*Database per service*" koncept. Slika ispod ilustruje jedan ovakav primer.



Ilustracija 11 – Primer sitema koji koristi posebnu bazu za svaki servis

Vrlo često se ovo odvija po nekom serverless principu tako da su određene baze aktivne samo kada je to potrebno. Na primer, u slučaju isplate zarada u nekoj kompaniji, koje se dešava jednom mesečno, formira se baza koja sadrži samo podatke neophodne za isplatu zarada i ona se aktivira samo jednom mesečno dok je ostatak vremena stopirana. U serverless smislu ovo znači da se resursi mnogo racionalnije troše i time se postiže značajna ušteda.

Postoje različite implementacije Cloud-Native baza podataka ali glavni koncept se odnosi na razdvajanje samih *podataka (storage)* od *computing (transakcije, cache, sql) sloja*. Ovakav pristup veoma dobro oslikava AWS Aurora - Relaciona baza podataka specijalno osmišljena od strane AWS-a za Cloud Native pristup.

U ovom slučaju, baze mogu biti i serverless. Obzirom da su podaci odvojeni, sam server baze se može skalirati po princupu *zero-to-max-and-back*. Ovo znači da se baza stopira i pokrene samo kada je potrebno. Na primer, ukoliko developer nešto testira i za to mu je potrebna baza, nije neophodno da je baza uvek dostupna već samo kada se prave upiti ka bazi.

Posebno je važno napraviti razliku između:

1. *Transakcionih* baza podataka

   Ovo su baze koje imaju *Online Transaction Processing (OLTP)* arhitekturu. Cilj ovakvih baza je da podrže real-time aplikacije. Fokus je na transkacijama (insert, update, delete, *veoma jednostavan select*) pri čemu se vodi računa da svi upiti budu precizni, brzi i da zahtevaju malo resursa. Pretrage su obično zasnovane na indeksima. Ovakva arhitektura je obično nezahvalna kada treba uraditi dublju analizu podataka selektovanjem i join-ovanjem više tabela.

2. *Analitičkih* baza podataka

   Ovo su baze koje imaju *Online Analytical Processing (OLTP)* arhitekturu. Ove baze obično imaju *Star* ili *Snowflake* šemu. Fokus je na čitanju podataka (select) korišćenjem *table scan-a* umesto indeksa.

Može se, sa određenom preciznošću, reći da su Transakcione baza podataka pogodnije za Cloud-Native pristup dok su Analitičke baze podataka pogodnije za tradicionalni pristup.

Neki najuniverzalniji i preporučeni pristup bi bio:

- Koristiti Transakcione baze kao Cloud-Native - više malih baza za svaki servis
- Koristiti Analitičke baze po tradicionalnom pristupu i shvatati ih kao *Data Warehouse.*
- Prevlačiti podatke iz Transakcionih u Analitičke baze samo kada je to potrebno, po principu *Extract, Transform, and Load (ETL)*

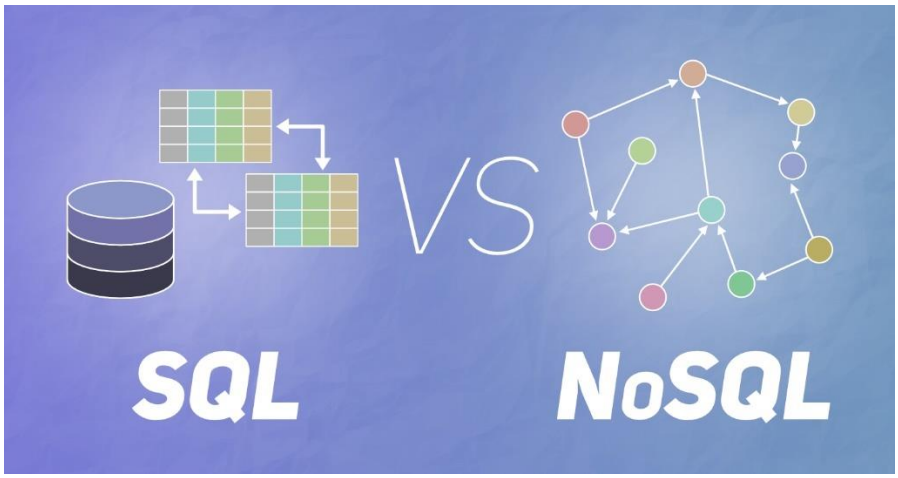
Jedan primer primene Cloud-Native pristupa za Analitičke baze bi bilo prethodno pomenuto skaliranje po principu zero-to-max-and-back. Baza se startuje onda kada je potrebno uraditi analizu podataka.

U principu, najviše prostora za primenu Cloud-Native principa postoji u slučaju Transakcionih baza podataka.

Kada govorimo o Transakcionim bazama, najčešće mislimo na Relacione baze kao što je MySQL, koje predstavljaju neki tradicionalni pristup. Ipak veoma je bitno pomenuti i *NoSQL* baze kao što je MongoDB. Ovo su baze koje ne koriste standardne strukture koje koriste relacione baze (kolone, redove, table, šeme) već se oslanjaju na na fleksibilnije modele podataka. Tipovi NoSQL baza su:

- *Key-value data stores*
- *Document stores*
- *Wide-column stores*
- *Graph stores*

Na slici ispod prikazana je, vrlo uprošćeno, razlika u strukturi relacionih (SQL) i NoSQL baza podataka.



Ilustracija 12 – Poređenje strukture SQL i NoSQL baza podataka

NoSQL baze su posebno korisne za skladištenje nestruktuiranih podataka i osmišljene su da odgovore na zahteve za performansama, skalabilnošću i fleksibilnošću Cloud-Native okruženja. Jednom rečju, NoSQL baze su mnogo elastičnije od tradicionalnih relacionih baza. Sama njihova unutrašnja arhitektura ih čini pogodnim za horizontalno skaliranje, klastering, geografsku dostupnost i slično.

NoSQL baze su posebno pogodne i za serverless princip. Veoma dobar primer ovakvog pristupa je *AWS Dynamo*.

Iako se ne primenjuje često u praksi, Cloud-Native baze se mogu deploy-ovati i na Kubernetes clusteru i to pomoću objekta *StatefulSet*.

6 Sloj otpornosti (Resilience layer)

Svi pomenuti servisi u nekim slučajevima mogu biti nedovoljni za upravljanje izuzetno velikim i diverzifikovanim sistemom mikroservisa. Premostiti vezu između nestalnih kontejnera i bezbednih i pouzdanih baza podataka često nije tako jednostavno, čak ni uz upotrenu svih do sada pomenutih metoda.

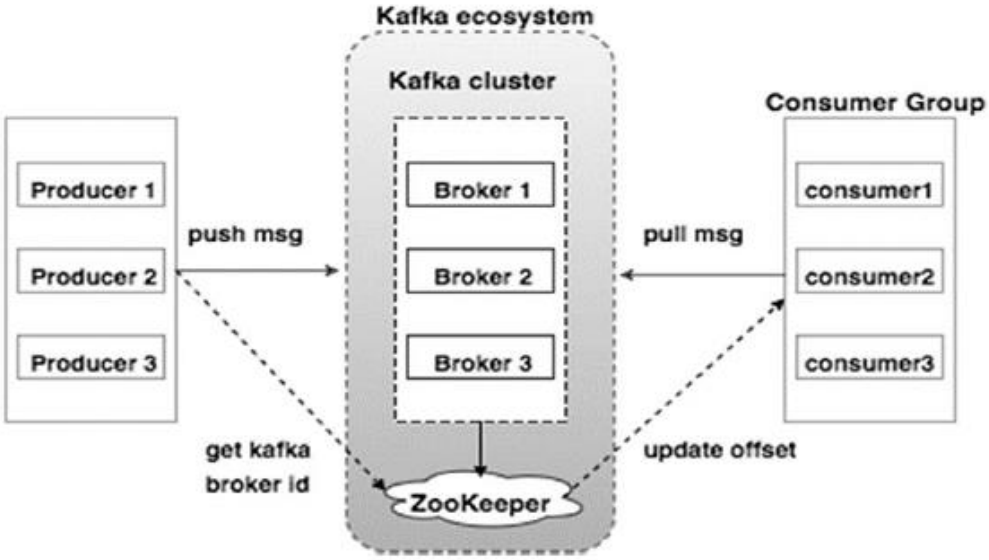
Jedno od veoma funkcionalnih rešenja za ovaj problem predstavlja tzv. *sloj otpornosti* **(***resilience layer***)**. Ovaj sloj se obično postavlja između kontejnera i baza podataka (ili drugih kontejnera) i predstavlja visoko pouzdani, privremeni storage podataka. Šta ovo znači za data pipeline:

- Servisi ne pričaju među sobom (ili sa bazama) direktno već indirektno, posredstvom sloja otpornosti.
- Podaci se u sloju otpornosti čuvaju na jedinstven način kako bi bili razumljivi svim zainteresovanim stranama. Očigledna prednost je to što nije neophodno trošiti mnogo vremena na direktno povezivanje servisa već se svi servisi, na jedistven i već ustanovljen način, integrišu samo sa slojem otpornosti.
- Uvodi se stabilnost sistema jer kompletan tok podataka i akcija (pipeline) ne funkcioniše po *push (guraj/šalji)* mehanizmu koji može opteretiti delove sistema i izazvati "crash" i gubitak podataka. Na primer. ako je sistem pod velikim opterećenjem, podaci se mogu prosto "pumpati" u sloj otpornosti a ostatak sistema će podatke preuzimati i obrađivati tempom koji odgovara dostupnim resursima.

Radi boljeg objašnjenja, uzećemo za primer *Apache Kafka*. Izvorno LinkedIN projekat, Apache Kafka je distribuirana streaming platforma koja ima 3 osnovne funkcije:

- *Publish (slanje/objava)* i *Subscribe (preuzimanje/pretplata)* poruka na *tokove* (*stream*-ove) *zapisa* (*topic*-e).
- Čuvanje stream-a *zapisa* (*topic*-a) na pouzdan način, otporan na greške i gubitak podataka uz vioku dostupnost (High Availability).
- Obrada stream-ova zapisa redom kojim se pojavljuju, kako bi se osigurala obrada svih podataka.

Arhitektura standardnog Kafka klastera ilustrovana je na slici na sledećoj strani.



Ilustracija 13 – Arhitektura Kafka klastera

Broker je osnovna komponenta koja čini Kafka klaster. Predstavlja instancu Kafke koja čuva podatke. Najčešće ih ima 2 ili više. Podaci se organizuju u Topic-e i dalje, prema particijama, grupama i faktoru replikacije, raspoređuju po svim dostupnim Brokerima unutar klastera.

Zookeeper je menadžment (management) komponenta koja služi za upravljanje Brokerima. Zookeeper sadrži informacije o različitim Topic-ima, na kojim Brokerima se nalaze, da li i gde imaju replike, kako su raspoređene particije, koji Broker je leader za neki Topic i slično. Zookeeper je obavezna komponenta bez koje Kafka klaster ne može da funkcioniše. Dovoljno je da postoji bar jedna Zookeeper instanca ali je preporučljivo imati ih više, faktički napraviti Zookeeper klister, kako bi postojala visoka dostupnost i ove komponente.

Consumer/Producer su spoljne aplikacije, ili delovi većih aplikacija, koje čitaju/pišu podatke iz ili u Topic-e.

Veoma bitan pojam, u smislu Kafka Sistema, je *Offset*, koji govori aplikaciji dokle je tačno stala prilikom čitanja Topic-a. Ovo je važno da bi svi podaci bili obrađeni i da ne bi bilo gubitaka.

Kafka se uvek vodi kao klaster a Topic-i se najčešće repliciraju na više Brokera, da bi u slučaju da Broker sa koga se trenutno čitaju podaci (Leader) "padne", tj. postane nedostupan, aplikacije mogle nesmetano da nastave da rade sa replikom istog Topic-a na nekom drugom Brokeru. U ovom slučaju, Zookeeper prosto proglasi neki drugi Broker kao Leader za posmatrani Topic.

Bitno je pomenuti da Kafka u potpunosti podržava cloud-native deployment model. Kafka kalster je moguće podići u okviru Kubernetes landscape-a, koristiti kao Cloud servis, i slično.

Neke od najčeščih oblika primene Kafke su:

- Kafka kao *Data Integration Bus*

  Omogućava jednostavno povezivanje nekoliko Producer-a sa velikim brojem Consumer-a.

- Kafka kao *Data Buffer*

  Postavljanjem Kafke ispred krajnjih odredišta podataka (baze podataka) dobija se veoma dobar bafer (buffer – privremeno skladište).

  Ovo je vrlo korisno ako je potrebno uraditi maintainance pozadinskog (backend) Sistema. U tom slučaju, podaci će se bez prekida ubacivati u Kafka klaster a kada sistem ponovo postane dostupan, samo će nastaviti da obrađuje podatke od momenta kada je zaustavljen, na osnovu Offset-a.

- Kafka kao *Short Time Data Storage*

  Nije neophodno, a često ni preporučljivo, čuvati sve podatke u Kafki. Najčešće se nad Topic-ima podešavaju *Retention* polise koje brišu podatke nakon određenog perioda ili određene količine zauzete memorije za skladištenje. Podaci se trajno čuvaju u bazama podataka pa se nakon obrade mogu slobodno obrisati iz Kafke.

Neke od alternativa za Apacke Kafku su:

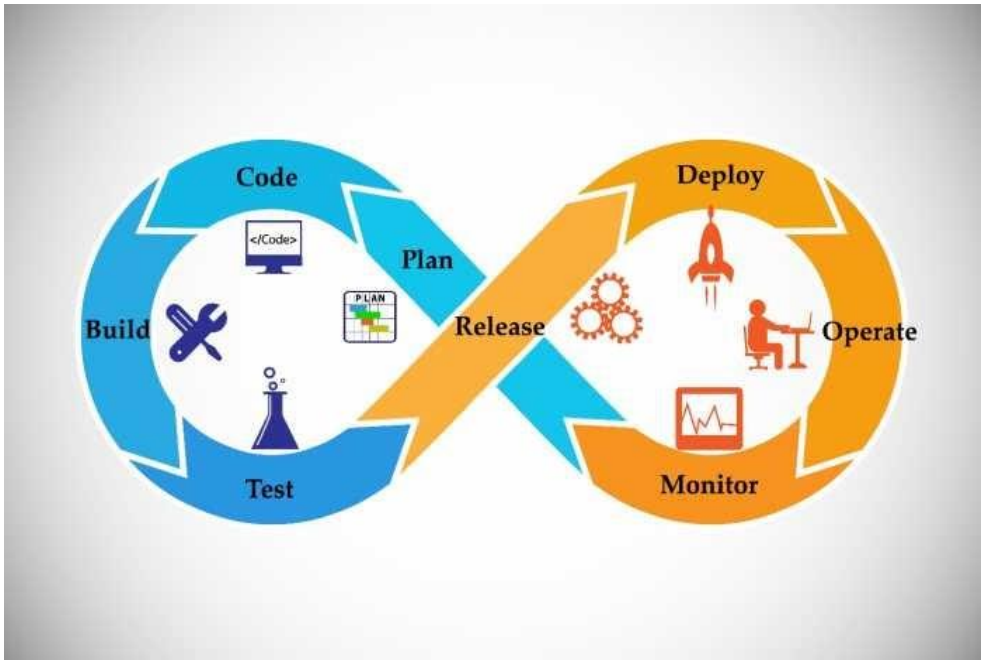
- *Rabbit MQ,*
- *Active MQ,*
- *Amazon Kine,*
- *Redis,*
- *Itd.*

7 DevOps koncept

DevOps metodologija ima za cilj da približi Development (Razvoj koda) i Operations (Operacije – upravljanje gotovim sistemom) radi postizanja bolje saradnje svih uključenih strana i efikasnijeg procesa od programiranja do dostave gotovog proizvoda i upravljanja i održavanja istog.

DevOps se može tumačiti na različite načine i različite kompanije, tj. timovi, mogu imati veoma različita shvatanja ovog koncepta. Ipak, većina akcija DevOps pristupa će uvek biti usmerena na automatizaciji uopšte kao i CI/CD proces.

Zapravo, DevOps se može shvatiti kao nastavak Agile development metoda. Korišćenjem različitih alata i procesa, obezbeđuje se brza i efikasna dostava softvera u više iteracija. Koraci jedne iteracije ilustrovani su na slici na sledećoj strani.



Ilustracija 14 – Koraci Agile development metoda

Kao što se može primetiti, fokus nije na predviđanju svakog mogućeg problema već što bržu dostavu softvera na realno okruženje, automatsko testiranje i praćenje, kao i brzo ispravljanje grešaka i ponovnu dostavu softvera uz započinjanje celog procesa ispočetka. Na ovaj način, softver se razvija *koninuirano* i na taj način se konstantno prilagođava okruženju, konstantno se rešavaju problemi ali i dodaju nove fukcionalnosti bez straha od destabilizacije ostalih komponenti sistema.

Ovakav pristup je mnogo prihvatljiviji i iz biznis perspektive. Naime, menadžmentu će biti mnogo prihvatljivija brzo gotova Demo verzija softvera, uz konstantne izmene i demonstracije u svakoj fazi razvoja koja prethodi produkciji, nego veoma obiman a pritom i veoma apstraktan plan razvoja tradicionalne aplikacije po Waterfall principu, koji je često veoma teško objasniti osobama koje nisu tehnički dobro potkovane.

Tehnički cilj DevOps pristupa u svakodnevnom radu bio bi da, posredstvom različitih alata i nivoa automatizacije, omogući Developerima da pišu, testiraju i debaguju kod sa što manje uključivanja Sistem Administratora. Neki od pristupa i rešenja za postizanje ovog cilja su:

- CI/CD pipeline
- Automatski monitoring i log collection svih aplikacija i sistema na jednom mestu, dostupno svima. Na ovo se nadovezuje i *alerting*, odnosno dojave u slučaju grešaka ili nestabilnosti operativnog Sistema.
- Automatizacija administratorskih taskova posredstvom alata koji na transparentan i bezbedan način obavljaju posao Sistem administratora, tako da ih Developeri mogu direktno koristiti a da ne ugroze celokupan sistem (npr. *ChatOps*)
- Racionalizacija korišćenja resursa bez promene u funkcionalnosti i performansama. Na primer, automatsko gašenje kompletnih Development okruženja van radnog vremena timova kao i automatsko podizanje (start up) istih sistema neposredno pre početka radnog vremena. Takođe je neophodno omogućiti alat kojim Developeri mogu sami podići (startovati) okruženje van radnog vremena, ukoliko se ukaže takva potreba (npr. ChatOps)

Obzirom da je više puta pomenut *ChatOps*, trebalo bi ga bliže objasniti. Naime, radi se o programiranju *ChatBot-a* koji na određene zahteve može, najčešće putem *WebHook-ova*, pokrenuti određene skripte, koje izvršavaju zadatke na infrastrukturi, i vratiti rezultat (output) u Chat kanal.

Ovaj pristup je često veoma dobro prihvaćen jer je interakcija sa ChatBot-om jednostavna, najčešće se može se obavljati putem chat alata koji tim svakako koristi (unutar posebnog chat kanala), sam po sebi predstavlja i Log i Monitoring ovakvih akcija jer svi korisnici mogu videti kada je neko zatražio neku akciju od ChatBot-a i koji je bio rezultat.

DevOps svakako učestvuje u svim aspektima planiranja i razvoja softvera jer daje odgovor kako na najbolji način iskoristiti dostupna okuženja i alate za razvoj, testiranje i pokretanje aplikacije. Kao što je ranije navedeno, planiranje aplikacije za Cloud-Native okruženje od početka je veoma važno tako da je input DevOps inžinjera u svim fazama razvoja veoma važan faktor.

8 CI/CD

CI/CD je skraćenica koja se odnosi na *Continuous Integration / Continuous Deployment (Delivery).*

Po definiciji, ovaj koncept predstavlja set operativnih principa i praksi koje omogućavaju development timovima da bolje sarađuju i dostavljaju novi kod mnogo češće i pouzdanije, eliminišući popularnu izreku "Radi na mom kompjuteru".

Continuous Integration ima tehnički cilj da uspostavi konzistentan i automatizovan način za verzionisanje, build-ovanje, pakovanje i testiranje aplikacija.

Continuous Deployment (Delivery) nastavlja tamo gde CI završava. CD automatizuje deployment (delivery) softvera na izabrano infrastrukturno okruženje.

Uprošćeno gledano, CI se postiže korišćenjem *sistema za verzionisanje koda (VCS)*, najčešće *GIT*. Ovaj pristup omogućava da više developera radi na istoj aplikaciji, bez preklapanja. Ceo proces može funkcionisati na veoma različite načine, zavisno od potreba tima i konkretnog projekta ali u nekoj osnovi funkcioniše na sledeći način:

- Za određeni projekat se napravi centralni repozitorijum.
- Odrede se osnovne *grane***,** npr. Dev, Test i Master.
- Developeri *kloniraju (git clone)* repozitorijum na svoje lokalne kompjutere, najčešće uzimajući Dev granu kao polaznu tačku.
- Svaki developer pravi lokalno sopstvenu granu na kojoj radi. Kao ime grane se obično uzima opis funkcionalnosti koja se programira ili broj tiketa pod kojim je zaveden zahtev.
- Nakon što završe posao, developeri čuvaju kod, uz komentare (*git commit*), i zatim šalju (*git push*) svoje izmene na centralni repozitorijum
- Pojedinačne grane se zatim, po nekoj ustanovljenoj proceduri, *spajaju (git merge)* sa glavnom (najčešće DEV) granom. Pritom se na propisan način rešavaju bilo kakvi *konflikti* koji su eventualno nastali ako su developeri imali preklapanje u pojedinačnim delovima koda koje su pisalil.
- Po pravilu, izmene se nikad ne šalju direktno na Test granu već se, nakon što se razreše konflikti na Dev grani, kompletne izmene prebacuju sa Dev na Test granu jednim merge zahtevom a na kraju na Master (produkcijska grana).

Nakon svakog commit-a na Dev, Test i Master grani, pokreće se CI/CD proces, poznat kao *Pipeline.*

Pipeline je niz automatiskih akcija koje build-uju, deploy-uju i testiraju aplikaciju. Opet, proces može funkcionisati na razne načine ali u nekoj osnovi:

- Na Dev grani se radi *white box testiranje***,** tj. testira se sam kod developera i proverava da li se izvršava sve što je predviđeno da se izvršava datim kodom (*unit testovi*)

- Na Test grani se radi *black box testiranje*, tj. testira se da li napisani kod odgovara korisničkim zahtevima (spoljašnje testiranje ili contract testovi). Ovde se eventualno uključuju realni podaci i dodatni dependency-ji koji postoje i na produkciji, kako bi se osiguralo da se aplikacija ponaša ispravno i sa realnim podacima. Takođe, ovde (ili u nekom među koraku) izvršavaju se i *Load* i *Stress testovi*.
- Poslednji deploy je na produkcijsko okruženje, za koji je najčešće vezana Master grana. Ovo je dakle poslednji korak koji napisani i testirani kod konačno stavlja u produkciju. Iako se sam proces odvija automatski ovom koraku se posvećuje posebna pažnja i najčešće se inicijalizuje od strane inžinjera, uz pomno praćenje celog procesa i detaljniji monitoring određeno vreme nakon deploymenta. Svakako na produkciji bi trebalo da postoji monitoring sistem koji će automatski alarmirati tim ako postoji neki problem.

Ponekad nije moguće zatvoriti kompletan krug od developmenta do produkcije sa CI/CD pipeline-om. Često se poslednji korak, deploy na produkciju, izvršava u potpuno odvojenom okruženju ili čak od strane klijenta. U ovom slučaju, nakon svih testova, pipeline builduje finalnu verziju aplikacije i pritom pravi *Artifact*. Ovo je poseban objekat koji predstavlja instalacioni paket aplikacije (u Cloud-Native svetu, ovo je najčešče Docker image uz određene skripte i šablone za deployment na orkestracioni sistem) koji se šalje inžinjeru koji ima pristup produkciji kako bi se manualno uradio deployment. Svakako, deployment se može uraditi i automatski, od strane drugog CI/CD sistema koji kao ulazni parametar uzima poslati Artifact.

Ovi procesi mogu biti prilično kompleksni. Moguće je npr. povezivati više pipeline-a, iz različitih projekata i buildovati jedan paket. Obzirom na kompleksnost ovih procesa i potrebom da se oni odvijaju veoma često, da bi mogli često izbacivati nove verzije aplikacije, zapravo je neophodno da se sve odvija na automatizovan način da bi uopšte bilo moguće raditi po ovom principu. Očigledno je da u suprotnom postoji veliki broj mogućnosti za ljudsku grešku.

Postoji veliki broj različitih alata koji obavljaju CI/CD proces: *Gitlab, Jenkins, TeamCity*, itd. Ovi alati mogu biti hostovani u okviru kompanije, u Cloud-u ili uzeti kao servis od nekog provajdera. Neka osnovna arhitektura koju prati većina alata je sledeća:

- *Centralni server* koji čuva repozitorijume koda, korisničke naloge, komentare, artifacte, itd.
- *Worker-i/Runner-i* - posebne mašine (serveri, VM, kontejneri, k8s podovi) koji pružaju izvršno okruženje neophodno za pojedinačne taskove svakog pipeline-a

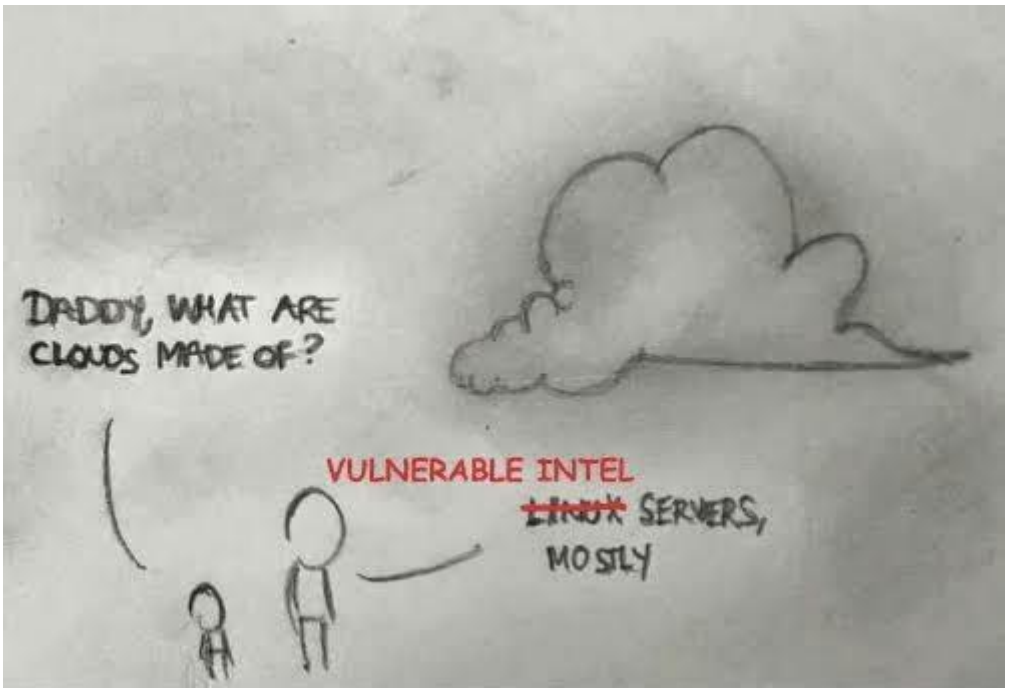
Ovakva podela je neophodna jer se ne mogu svi kodovi i projekti build-ovati na istom okruženju. Za java aplikacije potrebno je okruženje koje ima javu, za .Net aplikacije potreban je Windows Runner, i slično.

Kada su ovi sistemi hostovani u Cloud-u, ovde postoji mogućnost uštede u smislu da se runner instance aktiviraju samo kada postoje taskovi koji ih zahtevaju.

9 Aspekt bezbednosti

Do sada je većinom bilo reči o prednostima Cloud-Native pristupa, kao i Cloud-a uopšte. Veoma važna stavka bilo koje IT tehnologije ili pristupa je svakako bezbednost.

U neku ruku, može se reći da je bezbednost obrnuto proporcionalna dostupnosti informacija. Tj. što je određeni IT sistem korisniji i pristupačniji korisnicima (povezaniji sa drugim sistemima, veći, dinamičniji, skalabilniji, fleksibilniji, itd.) utoliko postoji više prostora za pojavu određenih slabosti koje mogu biti eksploatisane od treće strane.



Slika 3 – Šaljiva definicija bezbednosti klauda (cloud-a)

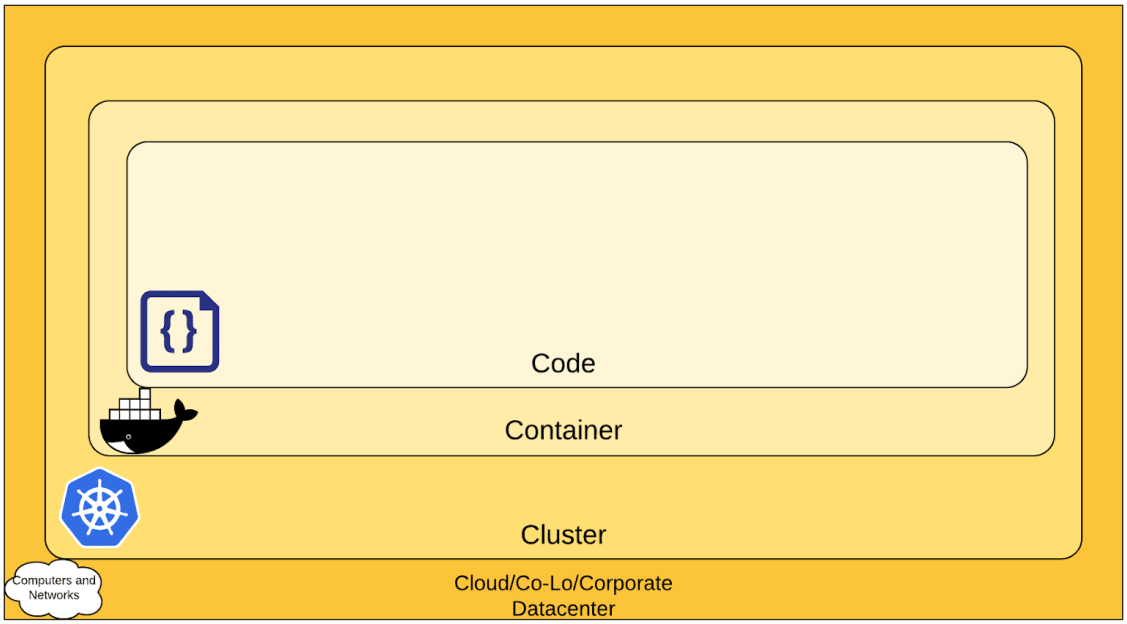
Dakle, obzirom da Cloud-Native rešava masu problema i otvara veliki broj novih mogućnosti, može se pretpostaviti da je neophodno utoliko više razmišljati o bezbednosti ovih sistema.

Neke osnovne karakteristike Cloud-Native sistema, kao što je već navedeno, su Cloud hosting i kontejnerizacija, što ih čini veoma elastičnim i skalabilnim. Očigledno je da, za razliku od tradicionalnih sistema gde je trebalo osigurati jednu veliku aplikaciju koja se nalazi na jednom mestu, ovde postoji ozbiljan izazov primene *distribuirane bezbednosti*, odnosno Cloud-Native bezbednosti.

Ovde je potrebno promeniti način na koji se posmatra bezbednost, odnosno nad kojim komponentama je potrebno primeniti i na koji način.

9.1 The 4 C's of Cloud-Native Security

Kubernetes definiše tzv. "*The 4 C's of Cloud-Native Security*", kao što je ilustrovano na slici ispod.



Ilustracija 15 – komponente koje čine Cloud-Native stack

Ovo je prilično dobar osnovni pregled koji važi za sve Cloud-Native tehnologije uopšte. Potrebno je posmatrati bezbednost ove 4 komponente.

Ono što je važno primetiti na ovoj slici jeste da je bezbednost viših slojeva direktno zavisna od bezbednosti nižih slojeva. Konkretno, bezbedan kod je beskoristan, u smislu bezbednosti, ukoliko postoje bezbednosni propusti u Kontejnerima, Klasteru, ili Cloud provajderu nad kojim se kod izvršava.

9.1.1 Cloud

Osnova svake Cloud tehnologije je sam Cloud, odnosno fizičke mašine (datacentri) i virtualizacija koja postoji kod Cloud provajdera. Obzirom da je ostatak Cloud-Native stack-a izgrađen sa Cloud okruženjem u osnovi, bezbednost na ovom nivou je polazna tačka od koje se mora krenuti da bismo mogli garantovati bezbednost celog sistema.

Cloud provajderi, obzirom na prirodu svog posla, vode računa o bezbednosti svojih komponenti ali ovo je ipak nešto što se veoma teško proverava i na šta klijent koji koristi Cloud usluge najčešće ne može mnogo da utiče. U slučaju Cloud-a, klijent dobija apstraktno okruženje i ne može sa sigurnošću proveriti da li Cloud provajder redovno patch-uje underlying OS.

U neku ruku, najsigurnija opcija je dobro se informisati, izabrati provajdera koji najbolje ispunjava zahteve sistema koji pravimo/održavamo kao i obezbediti dobar SLA

(Service Level Agreement) sa provajderom koji sa pravne strane garantuje dostupnost i kvalitet (kao i bezbednost) Cloud usluga.

Svakako, u slučaju kada su bezbednosni kriterijumi veoma visoki, postoji mogućnost da se koristi:

- Sopstveni datacentar
- Cloud usluge povezane sa sopstvenim data centrom
- Dedicated Cloud usluge - konkretan hardver koji Cloud provajder daje klijentu na ekskluzivno korišćenje

U ovom slučaju je bitno voditi računa o aspektu isplativosti (cost-benefit) jer su ove opcije po pravilu daleko skuplje od standardnih Cloud usluga.

Sa druge strane, korisnici mogu uticati na to kako oni koriste Cloud usluge. Bilo kakvi napori provajdera da osigura bezbednost biće bezuspešni ako korisnik sam ostavi otvoren port koji se može iskoristiti za upad. Provajderi obično daju vrlo detaljne smernice kako najbolje koristiti njihove usluge u smislu bezbednosti. Ovo su neke od njih:

Tabela 3 – Bezbednosne preporuke cloud provajdera

IaaS Provider	Link
Alibaba Cloud	https://www.alibabacloud.com/trust-center
Amazon Web Services	https://aws.amazon.com/security/
Google Cloud Platform	https://cloud.google.com/security/
IBM Cloud	https://www.ibm.com/cloud/security
Microsoft Azure	https://docs.microsoft.com/en-us/azure/security/azure-security
VMWare VSphere	https://www.vmware.com/security/hardening-guides.html

Izvor: https://kubernetes.io/docs/concepts/security/overview/#cloud-provider-security-table [posećeno dana: 02.01.2020.]

Provajderi često nude i specijalizovane usluge za osiguranje bezbednosti u vidu konsultanata, dedicated podrške, i slično.

Ovde je posebno važno napomenuti da je, osim bezbednosti hardver-a i OS-a, posebno potrebno obratiti pažnju na bezbednost na nivou mreže.

9.1.2 Cluster

Cluster je nešto što se možda pretežno odnosi na Kubernetes ali ipak, u svim Cloud-Native sistemima veoma često se govori o klasterima u nekom obliku.

Ovde se govori posebno o bezbednosti:

- *Komponenti koje čine klaster*
- *Komponenti koje rade unutar klastera*

Konkretne metode ojačavanja bezbednosti svakako zavise od toga o kakvom klasteru je reč. Svaka tehnologija je različita i sadrži različite komponente ali, slično kao za Cloud provajdere, za svaku tehnologiju postoje preporuke koje je potrebno proučiti i pratiti. Neke od opštih smernica o kojima treba voditi računa su:

- Autorizacija
- Autentifikacija
- Enkripcija
- Bezbednosne polise
- Kvalitet servisa i racionalno korišćenje resursa uz odgovarajuća ograničenja
- Mrežne polise

Bitno je napomenuti da je neophodno da ovo bude kontinuiran proces tako da se novi patch-evi i nadogradnje instaliraju redovno.

9.1.3 Container

Kao što je ranije pomenuto, kontejneri su osnova Cloud-Native tehnologija. Samim tim, neophodno je posvetiti posebnu pažnju bezbednosti u okviru kontejnera.

Možda najočiglednija slaba tačka su *imidži kontejnera* (*container images*). Neke od smernica su:

- Voditi računa da se image uzima *samo sa registara kojima se može verovati*. Ovo su najčešće privatni registri.
- Svakako, potrebno je posebnu pažnju posvetiti *bezbednosti samog registra* kroz enkripciju i autentifikaciju jer će bilo kakvi pokušaji da se osigura bezbednost na image nivou biti beznačajni ako se registar može lako kompromitovati.
- *Redovno skenirati i ažurirati image u okviru registra* - zastareli image može sadržati zastarele i nebezbedne aplikacije i biblioteke, pa čak i skriveni malware.
- Preporučljivo je i *potpisivati image digitalnom sertifikatom* i postaviti kontrole tako da samo ispravno potpisani image može ući u sistem. Na ovaj način se postiže *provera integriteta* i eliminišu napadi tipa *man-in-the-middle*. U slučaju Dockera, može se koristiti *Docker Content Trust (DCT)*.

Sledeća slaba tačka bi mogla biti *bezbednost orkestracije kontejnera,* odnosno alata koji se koriste za orkestraciju, *kao što je Kubernetes*. Ovo je veoma važan aspekt jer upravo je alat za orkestraciju komponenta koja sistemu daje dinamičnost, distribuiranost i elastičnost. Osim prednosti, orkestracija unosi i određenu kompleksnost tako da, ukoliko se ne postave striktna pravila funkcionisanja, dovoljni su čak i nepažljivi korisnici, koji čak ni nemaju loše namere, da izazovu razne probleme. Neke od preporuka su:

- *Obezbediti administrativni interfejs*
- *Odvojiti inter-cluster mrežni saobraćaj u diskretne virtualne mreže,* prema osetljivosti podataka koji se prenose
- Obezbediti dobru *izolaciju i ograničenja resursa* - jedna aplikacija je sme biti u mogućnosti da zauzme dovoljno resursa da ugrozi ostale aplikacije
- *End-to-end enkripcija i međusobna autentifikacija* inter-cluster mrežnog saobraćaja - posebno za komunikaciju između različitih worker nodova
- *Grupisanje kontejnera* istog ili sličnog bezbednosnog nivoa u okviru istih nodova

Svakako treba uzeti u obzir i same *kontejnere*. Neke smernice su:

- *Redovno instalirati zakrpe (patch)* i ažurirati container runtime. Određene greške u runtime-u se mogu iskoristiti za dobijanje pristupa svim ostalim kontejnerima na datom OS-u pa i samom OS-u.
- Treba posebno obratiti pažnju na *konfiguraciju kontejnera*. Neispravno konfigurisan kontejner može imati pristup ka previše uređaja, može imati mogućnost da pozive nebezbedne sistemske pozive, mount-uje osetljive direktorijume u read-write modu, pa čak i kompromituje sam OS na kome radi.
- *Onemogućiti privilegovane korisnike* unutar kontejnera

9.1.4 Code

Na kraju dolazimo do bezbednosti samog koda. Ovo je zapravo segment na koji najviše možemo da utičemo. *Bezbednost samog koda se ne razlikuje mnogo u tradicionalnom i Cloud-Native pristupu*. U osnovi, ako su ispoštovane sve bezbednoste preporuke na nižim slojevima i sve preporuke za deployment koda na Cloud-Native platformu, na ovom nivou se primenjuju tradicionalne bezbednosne preporuke. Neke od preporuka su:

- Svesti komunikaciju na TLS gde god je to moguće
- Ograničiti opseg portova za komunikaciju
- Osigurati bezbednost svih dependency-a koje naš kod može imati
- Sprovoditi redovno bezbednosno skeniranje koda
- Redovno testirati kod i voditi računa o pokrivenosti koda testovima
- Sprovoditi povremene *penentration testove*

Većina pomenutih preporuka se može automatizovati u okviru CI/CD pipeline-a. Automatizacija u svakom pogledu je veoma dobrodošla jer u suprotnom kontrola skalabilnog Cloud-Native sistema postaje praktično nemoguća.

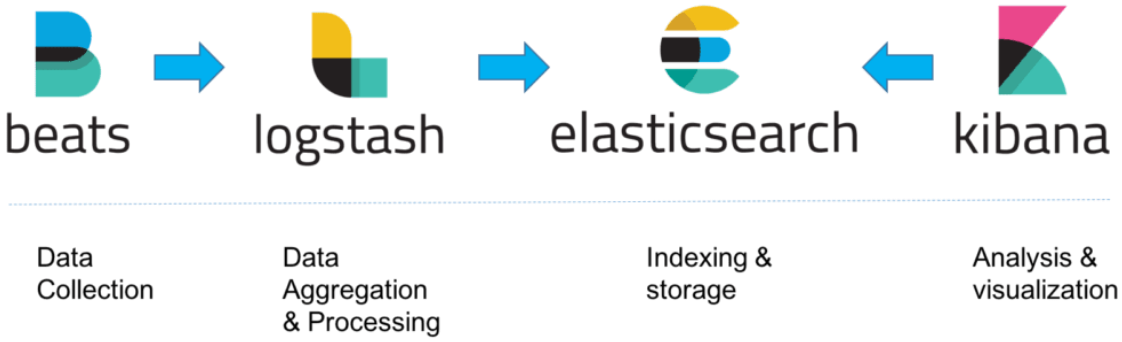
9.2 Detekcija – Logovi/Metrike i Monitoring

Sve do sada pomenuto odnosi se većinom na *prevenciju* narušavanja bezbednosni. Veoma bitan aspekt je i blagovremeno *otkrivanje* postojećih slabosti u sistemu koji je u upotrebi, kao i otkrivanje nastalih upada. Otkrivanje (detekcija) počinje prikupljanjem podataka *(logovi i metrike)* i njihovom analizom *(monitoring).*

Obzirom da aplikacije nisu na jednom mestu već distribuirane unutar kontejnera, klastera i klauda, nije tako jednostavno pretražiti logove ili sakupiti metrike performansi. Da bi se ovaj problem rešio, potrebni su posebni alati koji i sami imaju Cloud-Native arhitekturu.

9.2.1 Logovi

Uzećemo za primer ELK(B) stack. Ovo je open-source rešenje koje se sastoji iz nekoliko komponenti, čija imena zapravo čine anagram koji imenuje ovaj stack:

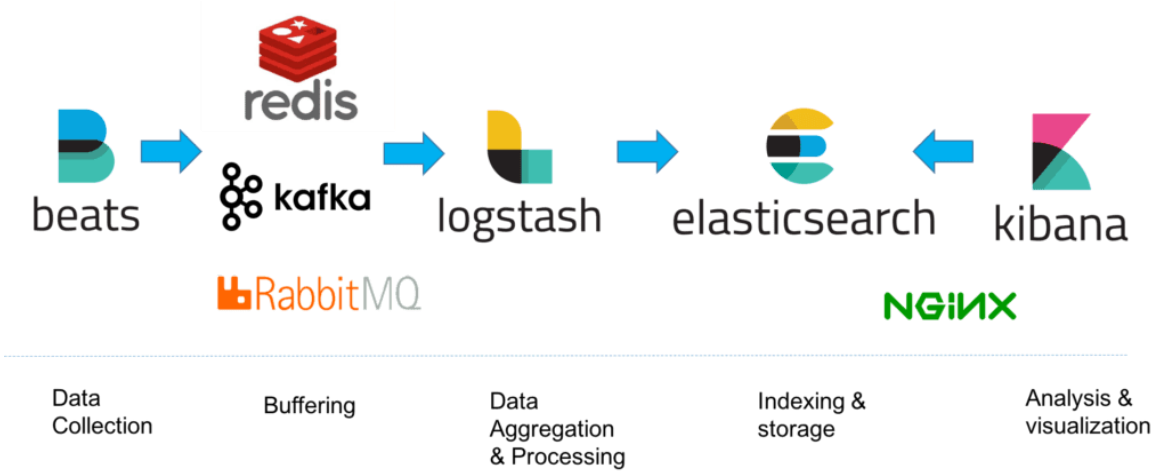


Ilustracija 16 – Komponente koje čine ELK(B) stack

Pipeline funkcioniše po principu:

- *Beats* je agent koji je distribuiran na sve sisteme i ima zadatak da prikuplja sve logove i šalje ih u logstash
- *Logstash* agregira i procesuira (parsira) podatke o logovima i šalje ih u elasticsearch
- *Elasticsearch* je NoSQL baza koja čuva sve podatke o logovima u obliku koji je pogodan za analizu
- *Kibana* vizuelno prikazuje podatke i omogućava pretragu i analizu

U ovaj standardni stack mogu se uključii dodatne komponente radi dobijanja otpornosti (npr. Kafka, kao što je ranije objašnjeno) i povećanje bezbednost samog logging sistema (Nginx):



Ilustracija 17 – ELK(B) stack sa dodatnim komponentama

Ovo je prilično dobar sistem jer prati skalabilnost datog Cloud-Native sistema čije logove posmatra i prikazuje na jednostavan način, što u velikoj meri olakšava bilo kakav debugging.

ELK(B) je izuzetno dobar primer Log Collection sistema. Svakako postoje i druga rešenja kao što su: *Splunk, Papertrail, Graylog, itd*. U suštini, sva Log Collection rešenja funkcionišu po relativno sličnom principu:

- *Agent za prikupljanje podataka*
- *Alat za procesuiranje podataka*
- *Storage*
- *Vizualizacija*

Razlika je samo u konkretnim alatima koji se koriste za obavljanje ovih funkcija, kao i konkretnih mogućnosti izabranih alata i njihovoj usklađenosti sa potrebama datog sistema.

U samom ELK(B) stacku određene komponente mogu u nekim slučajevima biti zamenjene drugim alatima. Takođe, komponente ELK(B) stack-a se mogu koristiti kao zasebna rešenja ili u kombinaciji sa drugim komponentama za potpuno drugačiju namenu.

Kada imamo sistem koji prikuplja logove, možemo imati i određeni monitoring nad tim logovima. Sistem za procesuiranje logova može npr. biti konfigurisan da u slučaju određenog tipa loga (Error) šalje notifikaciju (email, SMS, webhook, etc.) odgovornom timu.

9.2.2 Metrike

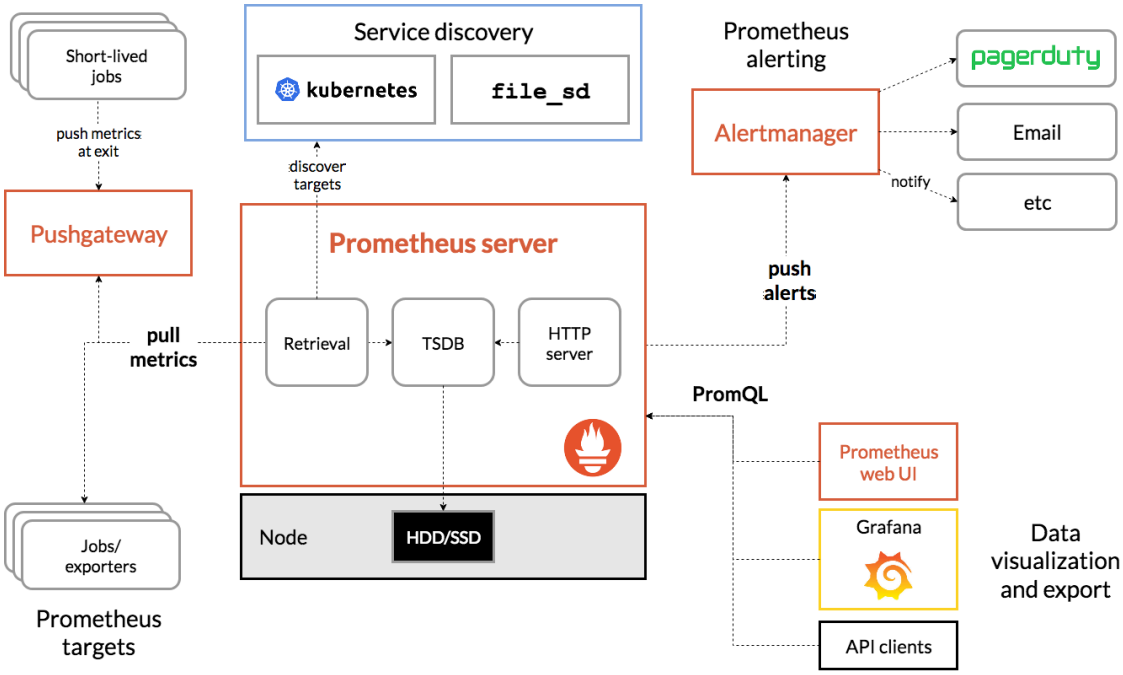
Osim logova, potrebno je posmatrati i performanse aplikacije kako bi se otkirle različite anomalije ili prosto ne-efikasan kod. LogCollection sistemi obično nisu pogodni i za prikupljanje metrika.

Dobar primer alata za prikupljanje metrika je *Prometheus*. Prometheus je open-source alat, specijalizovan za praćenje metrika aplikacija.

Prometheus se oslanja na veliki broj dostupnih exportera, izlistanih na stranici: https://prometheus.io/docs/instrumenting/exporters/ [posećeno dana: 03.01.2019.], da bi prikupljao podatke. Bitna činjenica ovde je to što se exporteri moraju implementirati na nivou aplikacije tj. koda (potrebno je planirati monitoring od početka pisanja koda).

Prometheus prikupljene podatke čuva u sopstvenoj bazi. Nad podacima se može vršiti analiza pomoću fleksibilnog query jezika pod nazivom *PromQL*. Uz Prometheus se obično koristi *Grafana*, za vizualizaciju podataka.

Arhitektura jednog sistema, sa Prometheus-om us osnovi, koji ima za cilj da prikuplja podatke o metrikama aplikacija u jednom Cloud-Native sistemu, analizira ih, prati i u određenim situacijama šalje notifikacije ilustrovana je na slici ispod.



Ilustracija 18 – Arhitektura Prometheus sistema za praćenje metrika

Ovakav sistem takođe prati skalabilnost Cloud-Native sistema i prikazuje podatke na jednostavan način, funkcionalno gledano slično kao prethodno objašnjeni LogCollection sistemi. Važna komponenta, koja omogućava praćenje skalabilnih aplikacija je *Service discovery*, koja omogućava da sistem sam registruje promene u broju instanci određene aplikacije, kao i uvođenje novih aplikacija u sistem.

Svakako, postoje i drugi alati, slični Prometheus-u: *Dynatrace Ruxit, netdata, DataDog, itd.*

10 Zaključna razmatranja

Na osnovu svega što je do sada napisano u ovom radu, Cloud-Native definitvno menja shvatanje aplikacija, sistema i IT-a kao celine.

Ono što treba posebno imati u vidu jeste da popularnost i opšta prihvaćenost bilo koje tehnologije ili pristupa nikako ne sme biti jedini razlog za upotrebu iste. Svaka situacija je jedinstvena i potrebe koje je neophodno zadovoljiti su retko nepromenljive. U svakom slučaju potrebno je objektivno sagledati šta je konkretan zadatak, kako se može rešiti i tek onda izabrati najbolji način za nastavak rada.

Postoji veliki broj prednosti i mana Cloud-Native pristupa, od kojih su mnoge navedene u ovom radu. Gledano u celosti, najveća prednost ovog pristupa, kao i najveći motiv za prihvatanje istog, je činjenica da je sam Cloud-Native izuzetno adaptilan, i usmeren na rešavanje problema kroz maksimalnu integraciju različitih alata, pristupa, sistema i aplikacija.

U nekom procesu digitalne transformacije, koji traje već duže vreme, kompjuteri su u neku ruku postali beskorisni ako nisu na mreži i ne mogu da komuniciraju sa drugim kompjuterima. Cloud-Native pristup, na neki način, predstavlja korak dalje i umanjuje razlike između slojeva dobro poznatog OSI referentnog modela, sa stanovišta pristupa dizajniranja IT sistema. U poslednje vreme, IT stručnjaci su sve manje specijalizovani za samo jednu oblast (mreže, OS, kod, hardver, i sl.). Pojavljuju se pojmovi kao što su “Cross functional teams”. U kompanijama u kojima se primenjuje DevOps pristup popularno se kaže “Svi smo mi DevOps-i”.

Činjenica je da je Cloud-Native danas neizostavni deo IT zajednice. U nekim situacijama je dobro primenjivati Cloud-Native principe a u nekim nije, ili je potrebno primenjivati u manjoj meri. Da bismo znali kada i kako da primenimo ovaj pristup, potrebno je da ga proučavamo u konstantno unapređujemo.

Cloud-Native je izuzetno velika oblast koju nije jednostavno shvatiti i može imati različito značenje u različitim situacijama za različite kompanije, timove i pojedince. Ovaj rad je pokušaj da bolje objasni Cloud-Native sa stanovišta ličnog iskustva autora. Kao što je već navedeno u uvodu, navode u ovom radu treba uzeti kao smernice i teme za razmišljnanje, a nikako kao striktna pravila. Ukoliko neko želi dalje da se bavi Cloud-Native temom, dobar početak je praćenje i proučavanje Cloud Native Computing Foundation: https://www.cncf.io/ [posećeno dana: 03.01.2019.]

Literatura

- https://www.cncf.io/ [dostupno dana: 26.12.2019.]
- Veinović, M. i Jevremović, A. (2018) *Računarske mreže*, Univerzitet Singidunum, Beograd
- https://about.gitlab.com/blog/2017/11/30/containers-kubernetes-basics/ [dostupno dana: 26.12.2019.]
- Veinović, M. i Jevremović, A. (2013) *Internet tehnologije*, Univerzitet Singidunum, Beograd
- https://docs.docker.com/ [dostupno dana: 30.12.2019.]
- https://containerjournal.com/topics/container-ecosystems/5-container-alternatives-to-docker/ [dostupno dana: 28.12.2019.]
- https://www.globaldots.com/blog/cloud-computing-types-of-cloud [dostupno dana: 30.12.2019.]
- https://kubernetes.io/docs/home/ [dostupno dana: 30.12.2019.]
- https://quarkus.io/ [dostupno dana: 30.12.2019.]
- https://www.quora.com/What-is-the-difference-between-transactional-data-and-analytical-data [dostupno dana: 02.01.2020.]
- https://riak.com/resources/nosql-databases/index.html?p=9937.html [dostupno dana: 02.01.2020.]
- https://upcloud.com/community/stories/databases-cloud-advantages-going-cloud-native/ [dostupno dana: 02.01.2020.]
- https://kafka.apache.org/ [dostupno dana: 03.01.2020.]
- https://kubernetes.io/docs/concepts/security/overview/ [dostupno dana: 03.01.2020.]
- https://techbeacon.com/enterprise-it/container-security-what-you-need-know-about-nist-standards [dostupno dana: 03.01.2020.]
- https://logz.io/learn/complete-guide-elk-stack/#intro [dostupno dana: 03.01.2020.]
- https://prometheus.io/docs/introduction/overview/ [dostupno dana: 03.01.2020.]