METODE DE UPGRADE IN KUBERNETES

Eusebiu Rizescu

Un aspect foarte important atunci cand o aplicatie este folosita de clienti este upgrade-ul acesteia la o versiune noua, cu un impact cat mai mic resimtit de utilizatori. Clientii se asteapta ca aplicatia sa fie disponibila tot timpul, iar dezvoltatorii de cod doresc sa faca upgrade-uri cat mai des pentru a testa si imbunatati cat mai repede noul cod. Exista mai multe strategii de upgrade, care vor fi discutate in acest document. Pentru validarea si testarea acestora, a fost simulat un upgrade de aplicatie in mai multe moduri, pe infrastructura de Kuberenetes.

An important aspect when an application is used by customers is that the upgrade of it to a new version, should have the least impact felt by users. Customers expect the application to be available at all times and developers want to upgrade as often as possible to test and improve the new code as soon as possible. There are several upgrade strategies that will be discussed in this document. To validate and test them, an application upgrade was simulated in several ways, on the Kuberenetes infrastructure.

Cuvinte cheie: Kubernetes, Upgrade, Downtime, Canary, Blue/Green, A/B Testing

1. Introducere

Kubernetes este un sistem de gestionare a containerelor, open-source, folosit pentru automatizarea implementarii, scalarii și distribuirii aplicatiilor. Acesta a fost inițial proiectat de Google si este acum un proiect intretinut de Cloud Native Computing Foundation . Scopul sau este de a oferi o "platformă de automatizare a implementarii, scalarii și operarii containerelor de aplicatii în grupuri de computere". Funcționează cu o gamă largă de unelte pentru containerizare, inclusiv cu Docker.

Desi actualizarea software-ului poate parea un proces simplu și direct, aplicațiile de azi sunt adanc integrate în funcții, procese și sisteme. Actualizarea software este inevitabila, fiecare aplicatie care este folosita si dezvoltata trebuie actualizata, fie ca vorbim de adaugarea de functii noi, fie ca vorbim de rezolvarea unor defecte deja existente. Inainte de fiecare punere in productie a unei aplicatii, trebuie analizat si dezvoltat un plan de upgrade. Nu toate aplicatiile sunt la fel, si astfel nu toate metodele de upgrade la fel. Trebui tinut cont de mai multe aspecte:

- Care este maximul de downtime pe care poate fi acceptat
- Care este maximul de resurse hardware care se pot adauga in timpul upgrade-ului
- Daca aplicatia poate rula pentru o perioada de timp cu 2 versiuni pornite in parallel (spre exemplu daca exista diferente de API care pot afecta utilizatorii finali)
- Daca aplicatia este stateful/stateless (daca este stateful, nu se poate pune versiunea noua instant, ci trebuie facut un upgrade treptat)

2. Metode de upgrade in Kubernetes

2.1 Recreate

Concept: Se sterg resursele corspunzatoare versiunii actuale si se creaza altele cu noua versiune

Resurse necesare in plus: Nu

Avantaje:

- Starea aplicatiei este reluata de la 0
- Nu este nevoie de resurse suplimentare
- Simplu de pus in practica

Dezavantaje:

- Downtime care depinde de oprirea/pornirea aplicatiei

Unde se poate folosi:

- In stadiul de dezvoltare a codului
- In aplicatii unde nu este foarte important downtime-ul
- In aplicatiile unde se face rar upgrade

2.2 Rolling Update

Concept: Aplicatie este inlocuita pas cu pas. Spre exemplu daca aplicatia are 3 poduri cu versiunea N. Se porneste un pod nou cu versiunea N+1. Se asteapta pana podul cu versiunea N+1 este functional. Se sterge un pod cu versiunea N. Apoi se repeta acest procedeu pana cand toate podurile au versiunea N+1.

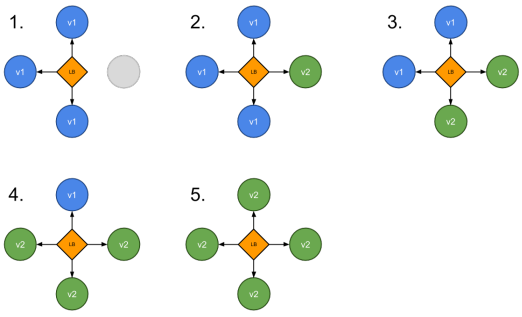


Fig. 1

Resurse necesare in plus: Da. Daca se doreste upgrade-ul secvential, atunci este nevoie de 1 pod in plus. Se poate bineinteles face si upgrade-ul mai multor poduri in paralel.

Avantaje:

- Versiunea noua este pusa live pas cu pas
- Folositoare in cazul aplicatiilor stateful care stiu sa isi rebalansese incarcarea

Dezavantaje:

- Necesita resurse suplimentare
- Upgrade/Rollback dureaza mai mult timp
- Neadecvata aplicatiilor care prezinta diferente de API-uri intre versiuni

Unde se poate folosi:

- Front-end-ul aplicatiilor
- Aplicatii unde faptul ca 2 versiuni pot coexista
- Cand nu este permisa existenta downtime-ului

2.3 Blue/Green

Concept: Inainte de aplicatia propriu zisa, exsita un router (load-balancer) care directioneaza traficul catre versiunea Blue sau Green. Versiunea N+1 (Green) este pornita fara ca versiunea N (blue) a fi stearsa. Dupa testarea faptului ca noua versiune indeplineste cerintele, router-ul incepe sa trimita trafic catre versiunea Green.

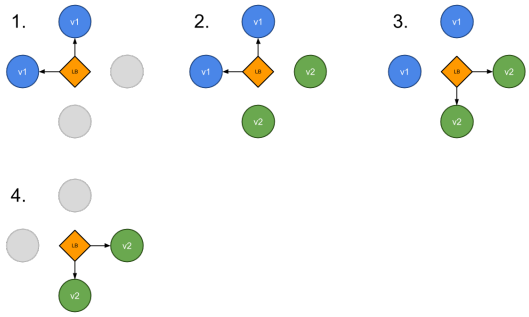


Fig. 2

Resurse necesare in plus: Da. Deoarece schimbarea traficului se face abrupt, versiunea Green trebuie sa fie complet scalata pentru a servi traficul curent. Astfel, avem nevoie de dublul resurselor.

Avantaje:

- Upgrade/Rollback instant
- Evita problemele de diferente intre versiuni, deoarece schimbarea versiunii aplicatiei in intreg clusterul se face instant
- Usor de implementat

Dezavantaje:

- Dublul resurselor necesar in plus
- Este nevoie de testare riguroasa pana ca o versiune sa ajunga in procesul de upgrade
- Pot aparea dificultati in cazul aplicatiilor stateful
- Tranzactiile curente pot fi afectate

Unde se poate folosi:

- Cand avem diferente de API-uri intre versiunile aplicatiei
- Resursele aplicatiei sunt mici si/sau bugetul este mare

2.4 Canary

Concept: Rutarea unui subset de utilizatori catre utilizarea noii versiuni (cu alte cuvinte se face testare direct in productie). Versiunea N+1 a aplicatiei este pornita, intr-o singura instanta, iar o parte mica din trafic este redirectata catre aceasta. Daca se observa ca pe acel subset de useri nu apar probleme, se mareste procentul, iar apoi versiunea canary se transforma in versiune de baza.

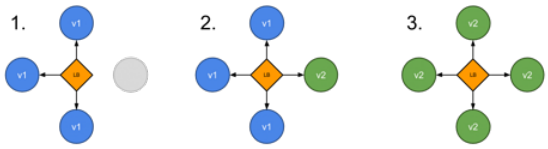


Fig. 3

Resurse necesare in plus: Da. Este necesar cel putin un pod in plus pentru noua versiune. Apoi se sterg poduri din versiunea veche si se adauga poduri cu versiunea noua.

Avantaje:

- Noua versiune ajunge la un subset de useri
- Folositoare pentru monitorizarea performantei si a ratei de eroare
- Rollback rapid
- Mai ieftina decat blue/green

Dezavantaje:

- Upgrade incet

Unde se poate folosi:

- Aplicatii unde faptul ca 2 versiuni pot coexista
- Cand nu este permisa existenta downtime-ului

2.5 A/B Testing

Concept: Aceasta tehnica este mai degraba o tehnica de business. Spre exemplu adaugarea unei functionalitati noi pentru un set de useri, sau modificare unei functionalitati pentru un set de useri si analiza ulterioara referitoare la care varianta este mai buna sau profitabila. A/B Testing este o extensie a Canary Deployment, in sensul ca subsetul de useri se poate calcula si pe trafic in sine, dar si pe anumite headere din request (User-Agent, headere speciale) precum si pe baza de cookies.

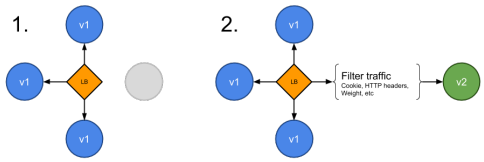


Fig. 4

Resurse suplimentare: Da. Este necesar cel putin un pod in plus pentru noua versiune. Apoi se sterg poduri din versiunea veche si se adauga poduri cu versiunea noua.

Avantaje:

- Control total asupra distributiei de traffic
- Rollback rapid
- Noua versiune ajunge la un subset de useri

Dezavantaje:

- Este nevoie de un loadbalancer inteligent
- Upgrade incet

Unde se poate folosi:

- Cand se doreste analiza unei modificari a unei functionalitati
- Front-end-ul aplicatiilor

3. Arhitectura folosita pentru testare

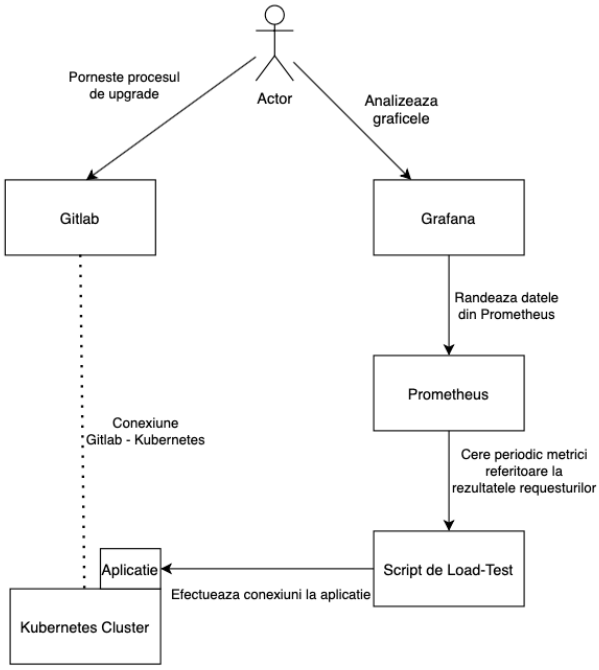


Fig. 5

3.1 Aplicatia

Aplicatia este una simpla, pentru testare, deoarece scopul lucrarii este de a testa diferite metode de upgrade, si nu a dezvolta o aplicatie complexa. Este o aplicatie web, dezvoltata in Flask (framework Python). Aceasta are doua versiuni, impachetate in 2 imagini Docker, si stocate pe DockerHub. Ambele versiuni returneaza pe calea „/” raspunsuri JSON de forma:

{"appVersion": "1", "status": "200"}

Bineinteles, versiunea 1 returneaza appVersion = 1 si versiunea 2 returneaza appVersion = 2. Pentru a avea control mai bun asupra analizei codurilor de raspuns, ambele versiuni de aplicatie returneaza mereu codul HTTP 200OK, insa un raspuns care contine status = 500 este considerat in analiza ca fiind eroare.

Aceasta aplicatie va rula pe clusterul de Kubernetes prezentat in sectiunea urmatoare.

Considerente pentru teste:

1. Pentru a simula un caz din realitate, ambele versiuni de aplicatie returneaza status = 500 pentru primele 60 de secunde de la pornirea aplicatiei, si astfel un pod nou, nu va fi pus in poolul de poduri sanatoase in primele 60 secunde. Ar fi neprofesionist a se pune in poolul de poduri active o aplicatie despre care se stie cu certitudine ca pentru o perioada de timp raspunde incorect.

2. Versiunea 1 returneaza mereu status = 200, adica este stabila pentru orice user agent.

3. Versiunea 2 returneaza pentru un singur user-agent, status = 500, cu o probabilitate configurabila din cod, iar pentru toti ceilalti user-agent returneaza status = 200. Note despre modul in care au fost configurate testele:
    - Versiunea 2 de aplicatie returneaza status = 500 pentru user-agent-ul „Sometimes-Fail”, cu probabilitatea de 50%
    - In load-test, se vor executa requesturi in numar egal pentru 2 useri agenti: „Always-Good” si „Sometimes-Fail”
    - Astfel, cu considerentele de mai sus, daca 100% din trafic va fi pe versiunea 2, vom avea 75% din trafic considerat corect si 25% din trafic considerat cu eroare.
4. Versiunea 1 va rula pe 3 pod-uri. Versiunea 2 va rula pe numar variabil de poduri in functie de metoda de upgrade.

3.2 Clusterul de Kubernetes

Clusterul de Kubernetes ruleaza cu ajutorul Minikube, pe masina locala, adica intr-un singur nod. In productie, acest cluster este necesar sa ruleze pe mai multe clustere pentru a asigura toleranta la defecte hardware sau software la nivelul serverelor. In testele facute, aici ruleaza aplicatia cu componentele sale. Cu acest cluster interactioneaza Gitlab care il instructioneaza prin API ce resurse sa porneasca/modifice/opreasca. Totodata, scriptul de load test, trimite permanent requesturi la aplicatia ce ruleaza pe acest cluster, pentru a analiza codurile de raspuns.

3.3 Gitlab

In Gitlab sunt tinute atat manifesturile de Kubernetes care descriu resursele si infrastructura aplicatiei, cat si pipeline-ul de creare, upgrade, stergere. El interactioneza cu clusterul de Kubernetes prin ajutorul Kubernetes API. Pentru conectare, de pe routerul local s-a facut forwardare de porturi catre masina locala unde ruleaza Kubernetes si realizata conexiunea cu accest cluster (pe baza de certificat si token), pentru ca mai apoi runneri din Gitlab.com sa poata executa comenzi pe acest cluster.

Datorita conexiunii dintre Gitlab si clusterul de Kubernetes, putem vedea in interfata de Gitlab statusul actual al clusterului (cate poduri avem pornite, starea lor). In plus, in cazul Canary, putem seta din interfata procentul de trafic pe care il dorim catre noua versiune de aplictie:

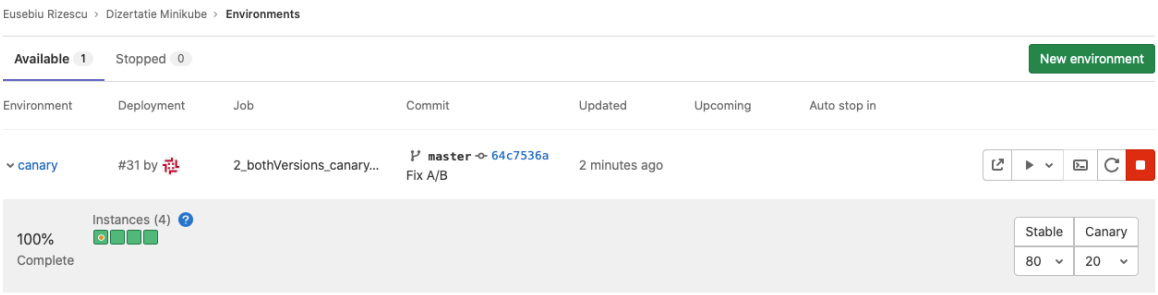


Fig. 6

Astfel, direct din interfata de Gitlab, putem vedea cate poduri de aplicatie avem (3 cuburi versi) si cate poduri de versiune Canary avem (1 cub verde cu bulina portocalie).

Pipeline-ul din Gitlab este urmatorul

- Pasul 0: Apeleaza pasii: pasul 4 → asteapta 60s → pasul 1 → asteapta 300s → pasul 2 → asteapta 300s → pasul 3 → asteapta 300s → pasul 4
- Pasul 1: Porneste resursele necesare pentru a livra Versiunea 1
- Pasul 2: Porneste resursele necesare pentru a livra si Versiunea 2
- Pasul 3: Sterge Versiunea 2 pentru a livra doar Versiunea 1
- Pasul 4: Sterge toate resursele din namespace

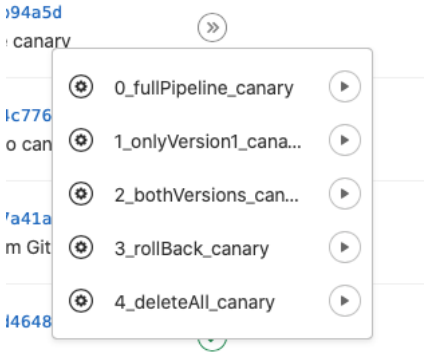


Fig. 7

3.4 Scriptul de load-test

Acest script este dezvoltat in Python 3 si trimite permanent (cu pauza de 0.01 secunde) requesturi catre aplicatie. Acest script, trimite requesturi in numar egal folosind 2 useri agenti: „Always-Good” si „Sometimes-Fail”, pentru a simula trafic din realitate. Bineinteles, s-ar fi putut folosi 100 de user-agenti, dar pentru a se observa mai usor pe grafice diferentele intre modalitatile de deploy, am ales numere mici (2 useri agenti) si care sa se imparta frumos (rata de fail 50% pe versiunea 2) .

Cum a fost precizat mai sus, ambele versiuni de aplicatie raspund cu codul HTTP 200OK, insa pentru a determina daca raspunsul este interpretat ca fiind corect sau eroare, vom analiza JSON-ul primit ca raspuns. Scriptul va analiza fiecare raspuns si va forma 4 metrici: „requests200V1”, „requests500V1”, „requests200V2”, „requests500V2”.

Problema care apare este aceea de stocare a acestor metrici in functie de timp. Astfle, acest script de load-test expune si metrici (similar cu node-exporter), care vor fi ulterior citite periodic (fiecare 5 secunde), de catre Prometheus. Requesturile permanente ruleaza pe un thread separat si formeaza aceste metrici, iar, cu ajutorul framework-ului Flask, scriptul le expune pe calea „/metrics”. Avand in vedere ca aceste metrici sunt crescatoare (countere), la fiecare request al Prometheus-ului catre scriptul de load test, metricile vor fi resetate. Cu aceasta abordare, in Prometheus vom avea metrici de forma „sum(requests) per 5 minute”.

3.5 Prometheus

Pentru a stoca istoria statusurilor requesturilor initiate de scriptul de load-test, se va folosi o instanta de Prometheus. Aceasta este configurata ca la fiecare 5 secunde sa colecteze metrici de la scriptul de load-test, pe care le stocheaza in baza lui de date interna.

3.6 Grafana

Deoarece Prometheus nu exceleaza la capitolul de vizualizare a metricilor, s-a folosit Grafana, care are ca sursa de date Prometheus-ul configurat anterior. In Grafana s-a creat un dashboard care contine mai multe grafice care ne pot ajuta la compararea metodelor de deploy. Acestea sunt:

1. Distributia de trafic (cat % versiunea 1 si cat % versiunea 2)
2. Rata de eroare versiunea 2
3. Rata de eroare total
4. Numarul de requesturi: V1 200, V1 500, V2 200, V2 500. (V1 500 nu este cazul deoarece avem mereu 200 pe versiunea 1).

Fig. 8

4. Rezultatele testelor

Metoda/Parametru	Downtime la upgrade	Rata de eroare dupa upgrade	Downtine la downgrade	Resurse in plus necesare
Recreate	Da	25%	Da	Nu
Rolling update	Nu	8%	Nu	+33%, podul cu versiunea noua
Blue/Green	Nu	25%	Nu	+100%
Canary (20%)	Nu	5%	Nu	+33%, podul pentru canary
A/B testing (Always-Good pe Versiunea 2)	Nu	25%	Nu	+100%
A/B testing (Sometimes-Fail pe Versiunea 2)	Nu	0%	Nu	+100%

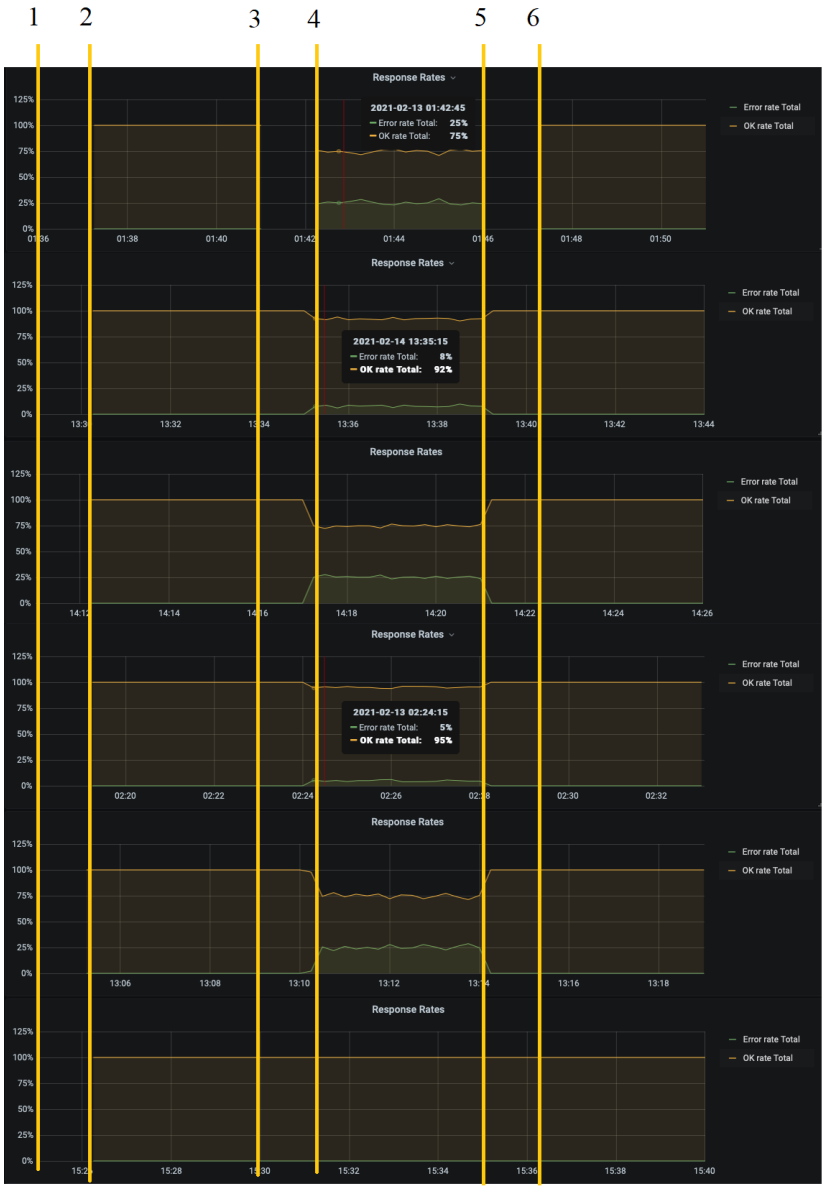
Pentru fiecare varianta de deploy testele au constat in:

- Namespace-ul de Kuberenetes nu contine nici o resursa
- Se porneste versiunea 1
- Nu se intervine 300 secunde
- Se lanseaza versiunea 2
- Nu se intervine 300 secunde
- Se face rollback la versiunea 1

Reamintim cum este configurata aplicatia:

- Un pod nou de aplicatie, in primele 60 secunde nu servceste requesturi
- Versiunea 1 raspunde numai cu OK
- Versiunea 2 raspunde pentru user-agent-ul „Sometimes-Fail” cu 50% probabilitate ERROR
- Aplicatia ruleaza pe 3 poduri (acest lucru conteaza la calcularea procentului de ERROR)
- Scriptul de load-test lanseaza requesturi cu 2 useri agenti: „Always-Good” si „Sometimes-Fail” in proportii egale

In figura de mai jos, se poate observa cum se comporta fiecare metoda de upgrade:



Recreate
Downtime
25% error rate

Rolling
No Downtime
8% error rate

Blue/Green
No Downtime
25% error rate

Canary
No Downtime
5% error rate

A/B Testing with
"Sometimes-Fail"
on Version 2
No Downtime
25% error rate

A/B Testing with
"Always-Good"
on Version 2
No Downtime
0% error rate

Timpi:
1 (t0) - Pornire versiunea 1
2 (t0 + 60s) - Versiunea 1 este live
3 (t0 + 300s) - Upgrade la versiunea 2
4 (t0 + 360s) - Versiunea 2 este live
5 (t0 + 600s) - Rollback (in afara de Recreate, la toate celealte trecera se face instant)
6 (t0 + 660s) - Versiunea 1 este live (la Recreate)

Fig. 9

5. Concluzii

Din pacate, nu exista o solutie de upgrade care sa se potriveasca fiecarei aplicatii. Este necesar ca inainte de a alege o varianta, aceasta sa fie inteleasa foarte bine, dar si sa se analizeze si celelalte variante existente. In plus, este important ca echipele de developeri si de operations sa lucreze impreuna pentru a alege varianta corecta pentru aplicatia in cauza. In acest document au fost prezentate si testate diferite tehnici individual, dar acestea pot fi combinate in functie de cerinte, pentru a obtine o solutie adecvata aplicatiei si infrastructurii existente.

B I B L I O G R A F I E

[1] Continuous Delivery - Reliable Software Releases Through Build, Test And Deployment Automation – Humble J., Farley D.

[2] https://kubernetes.io

[3] https://blog.container-solutions.com/kubernetes-deployment-strategies

[4] https://medium.com/google-cloud/kubernetes-canary-deployments-for-mere-mortals-13728ce032fe