Survey and Evaluation of Blue-Green Deployment Techniques in Cloud Native Environments

Bo Yang[1(✉)], Anca Sailer[2], and Ajay Mohindra[3]

[1] IBM Research – China, IBM, Beijing, China
yangbbo@cn.ibm.com
[2] IBM Research, IBM, New York, USA
ancas@us.ibm.com
[3] IBM Watson Health, IBM, New York, USA
ajaym@us.ibm.com

Abstract. Today, the cloud computing customers assume that the services or applications consumed from the cloud are always on, highly available for uninterrupted utilization. The requirement then for the service providers becomes to minimize the planned maintenance windows duration in order to reduce their repercussions on the service availability for the consumers. We evaluate in this paper the continuous deployment methodology called Blue/Green deployment which aims to support zero maintenance windows, and consequently to avoid any interruption to the end users. Our experiments analyze the most common Blue/Green deployment techniques in the industry, measure and normalize their behavior, and aim to identify the approach with the best performing continuous delivery as compared to the available technologies.

Keywords: Continuous delivery · Blue/Green deployment · High availability · Service discovery

1 Introduction

The Blue/Green deployment technology provides support for DevOps continuous delivery [1–3] with zero or near zero-downtime. This technology uses two different environments hosting two different versions of the service. The goal is to shift the incoming traffic from the environment hosting the current service version to the environment hosting the new service version. In most implementations, only one of the environments is live and thus serving all the production traffic. The live environment is typically considered "Blue", while the idle "to be" production environment is called "Green", as shown in Fig. 1. The key challenge of the Blue/Green deployment is the cut-over phase, when taking the service from its Green final stage of testing to Blue to handle the live production traffic. The zero or near-zero maintenance downtime comes down to how efficient the Blue/Green switch is performed.

In this paper, we first evaluate the state-of-art and current practices, and report on key performance results identified. We summarize in Sect. 2 the most prevalent Blue/Green deployment techniques and detail in Sect. 3 two example implementations

S. Yangui et al. (Eds.): ICSOC 2019 Workshops, LNCS 12019, pp. 69–81, 2020.
https://doi.org/10.1007/978-3-030-45989-5_6

of Blue/Green based on service discovery framework which is the most advanced technique. Section 4 presents the outline of our experiments for each implementation of Blue/Green deployment described in Sect. 2 and detailed in Sect. 3. We describe in Sect. 5 the experimental results and their analysis, pointing out the features and relevant scenarios for each implementation of the Blue/Green deployment techniques investigated. Finally, we summarize the paper and discuss the challenges and potential future work in Sect. 6.

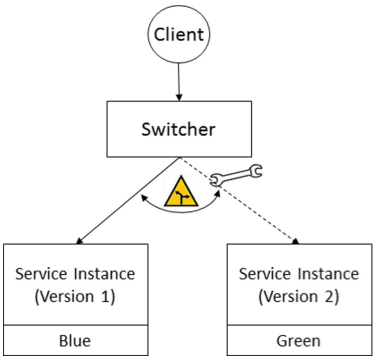

Fig. 1. Overall architecture of migration system with compliance validation

2 Blue/Green Deployment Related Work

The Blue/Green (BG) terminology was first introduces by Fowler [4] and it is just a way to distinguish between the two separate environments hosting the current (Blue) and new (Green) service releases. Other references call it A/B deployment [5], or Red/Black [6] deployment. Although there are slight differences between their overall goals of the upgrade, the common denominator is the key challenge of switching efficiently between the two environments. The various techniques that implement the switch from the service's Blue current version to its Green new version, impact in specific ways the performance of the B/G deployment. Before detailing the metrics we considered for comparing the various existing implementations, we summarize below the most prevalent implementations of B/G deployment.

2.1 Domain Name System (DNS)

These techniques rely on the DNS record update for the B/G switch and thus can be implemented with any of the leading DNS service providers, such as Cloudflare [7], DigitalOcean [8], Google Cloud DNS [9]. For example in Tutum and Cloudflare based implementation [10], CloudFlare cli is used to edit DNS CNAME entry for the B/G switch. The Amazon DNS based B/G technique is the DNS Routing Update with Amazon Route 53 [11]. This technique applies to single instances switch, swapping the environment of an Elastic Beanstalk application, cloning a stack in AWS OpsWorks and updating DNS with alternative environment's IP address [11].

2.2 Software Reconfiguration

These techniques rely on software reconfiguration for the B/G switch. Cloud Foundry (CF) leverages a CF Router [12]. Once a new service release is ready for production traffic, the CF Router is updated to remap the route to the new release. Virtual IP based solutions such as Floating IP [13] in Digital Ocean and Elastic IP [14] in AWS are used for single node.

B/G switch, where in the association of the virtual IP is changed. The AWS techniques which fall into this category are swapping the Auto Scaling group behind Elastic Load Balancer and updating Auto Scaling Group launch configurations [14]. This technique is not as granular as the DNS technique, but the traffic switch is more efficient.

2.3 Load Balancer

These techniques leverage a load balancer to trigger the change of routing configuration. For example, IBM Urban Code Deploy [15] works with Blue and Green environments hosted on the same machines, but different ports. The switching is achieved by changing the port in the load balancer routing rules. Examples of load balancers leveraged in this type of B/G deployments are HAProxy and nginx. The B/G deployment technique for docker uses nginx [16, 17], where nginx runtime configuration reload feature is used for the B/G switch. In the B/G deployment with HAProxy [18], the HAProxy health check is used for the B/G switch.

2.4 Service Discovery

These implementations use a level 7 service discovery framework to switch to a new service release. The service discovery is the automatic detection of services offered in an environment. One such framework example is the Netflix Eureka service discovery [19] which works together with the Zuul dynamic routing [20] to support zero-downtime rolling deployments [21]. Another example is Kubernetes [22] and the ISTIO intelligent router [23] which allows to configure service-level properties like circuit breakers, timeouts, and retries, for B/G deployments as detailed in the next section.

3 Blue/Green Deployment with Service Discovery

To route a request to its destination, we need to know the network location (IP address and port) of the targeted service instance. In a traditional application the network locations of the service instances are relatively static and could be retrieved from a configuration file that is occasionally updated. In a cloud native environment, however, this is a much more difficult problem to solve since the service instances have dynamically assigned network locations and the instance itself changes dynamically because of auto-scaling, migration, and upgrades. Thus, in most implementations of B/G deployment there are two main challenges: (1) the routing rules update requires

additional efforts to collect the new service instances IP hosts information, particularly in a dynamic auto-scaling environment; (2) the routing rules update on the router/load balancer service takes a significant amount of time to become effective because of cache on each node in routing path, thus affecting the service's version overlap or availability. To address these challenges, solutions like Netflix and Kubernetes use service discovery-based solutions (Zuul and Eureka [19, 20], ISTIO [23]) for VM and container deployments, as illustrated in Fig. 2.

In this paper, we use a test service named hereafter "My-Service", which registers its release version 1 for Blue and version 2 for Green. It uses node 1 and node 2 for the current, Blue service instance and node 3 and node 4 for the new, Green service instance. The incoming traffic reaches our Blue environment via a secure gateway, e.g., IBM DataPower for VMs and Kubernetes Ingress for the Kube cluster, which validates the applications calls credentials for My-Service and routes the calls to the dynamic router. My-Service is deployed on a cluster with multiple nodes. In order to support the automated service lifecycle management, and hence automatic deployment and switching of release versions, we leverage automation pipelines such as IBM Urban Code Deploy (UCD) [15] and IBM Cloud Delivery Pipeline [24]. In the pipeline, all the deployment locations are managed through the cloud management API, e.g., SoftLayer API for VMs and Kubectl API for Kubernetes. We first deploy two service instances in the target environments (Blue and Green), then we trigger the version switching process by invoking the pipeline as detailed hereafter.

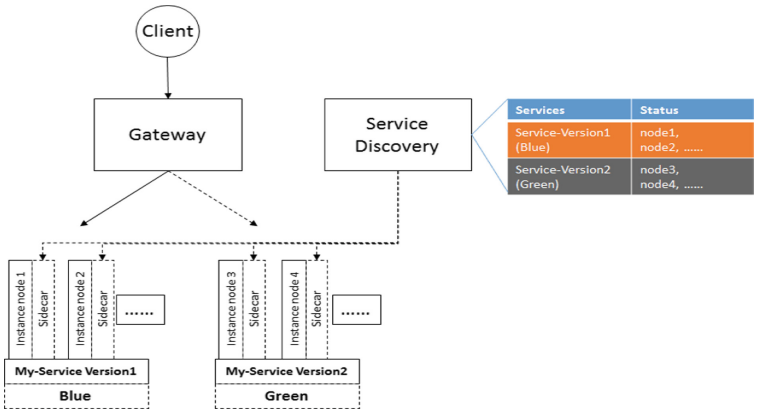

Fig. 2. Implementation for service discovery based Blue/Green Deployment. (Color figure online)

3.1 Blue/Green Deployment with Eureka and Zuul

We implemented this technique for VM based environments where a node in Fig. 2 indicates a VM hosting a service instance. Zuul, as the dynamic router, queries the service registry, Eureka, by using the service name from each incoming call, to retrieve the actual Blue IP hosts where to load balance the calls. To enable My-services' nodes

to register with Eureka, Netflix uses a sidecar agent on each node to communicate with the service registry server. Hence, we need to install the sidecar agent on each node and manage the sidecar's configuration file to detect the service registry server, register itself and start sending the health check heartbeats necessary to preserve the registration.

Zuul is continuing to route the traffic to those nodes tagged "My-Service-BLUE-ENV" in Eureka, during the backend nodes being replaced with those installed with the new service version. For the end user, this is a black box of traffic shifting from the old service version to the new one. The traffic switch is controlled via a pipeline orchestration tool (e.g. UCD or Jenkins). This approach does not require load balancer rule updates, nor DNS updates, nor router reboot, aiming for a real zero downtime switch.

3.2 Blue/Green Deployment with Kubernetes and ISTIO

We implemented this technique for the container based deployments. The container cluster environment is set up using the IBM Cloud Kubernetes service. In this case, a node in Fig. 2 indicates a pod which runs a service instance. ISTIO is the dynamic router that parses the uri of each incoming call for the service name, queries based on the service name to retrieve the hosts registered for the service and routes then the call to the retrieved Blue hosts load balancing the calls. To enable My-Service' pod to register with ISTIO, an istio-sidecar is used on each pod to communicate with ISTIO. Hence, we need to manage the pod deployment configuration to make it inject the istio-sidecar with the service instance and register itself to ISTIO. All the requests to the targeted service will be routed by ISTIO according to the predefined routing policies.

Similar to the automation for VM based environment, we leveraged the IBM Cloud Delivery Pipeline to enable the automation for service instance deployment and traffic routing configuration update.

4 Blue/Green Experimental Setup and Evaluation Metrics

4.1 Experimental Setup

We implement with Node.js for My-Service as test service API which returns when called its version information, and deploy it in two identical environments as instances of the service configured with different version information, i.e., "version1" for Blue and "version2" for Green. Moreover, we use a Switcher to shift the request traffic from one service instance (Blue) to the other (Green). In this paper, we implement the Switcher using five B/G deployment techniques: (1) AWS R53 DNS [11], (2) AWS Load Balancer-Auto Scaling Group (LB-ASG) [11], (3) Cloud Foundry Route Remapping (CF-RR) [12], (4) Netflix Service Discovery (NSD) based solution, and (5) Kubernetes Service Discovery (KSD) based solution. We also implement and deploy a Tester which sends curl requests every second and records the response routed from the Switcher. The response includes the request time, the response time, and the replied version information for each request.

To evaluate the selected Blue/Green deployment techniques performance, we setup five experiment environments for My-Service deployments as described in following.

For the AWS R53 DNS based B/G deployment [11], we created two EC2 instances on AWS for service deployment, one for Blue and another one for Green., and configured a DNS (e.g., bgtest.res-lab.ibm.biz) with the public IP of the Blue EC2 instance.

For the AWS LB-ASG based B/G deployment [11], we created two AMI images with two different versions of service. And we also created a Launch Template, an auto scaling group (ASG) instance and a load balancer (LB) instance as required to work with the ASG instance to route the request at the unique (LB) access endpoint when the backend EC2 instance is changed.

For the CF-RR based B/G deployment [12] on a container-based architecture, we published two Node.js applications (Blue and Green) on Cloud Foundry with two different versions of the service, and developed a switch script on the client side using the CF CLI to trigger the route remapping.

For the Eureka and Zuul based B/G deployment, we created two VMs on SoftLayer, one serving as Zuul server, and another one serving as Eureka server. We also created two more VM, deployed the Eureka sidecar on each of them, and then deployed the service with two different versions on each VM. We developed an UCD process to manage the sidecar's configuration and operation as described in Sect. 3.2, including the "update registration" process.

For the Kube and ISTIO based B/G deployment, we deployed ISTIO (v1.0) within istio namespace (it is used to isolate and manage a set of resource group in Kube) in Kubernetes, and created another two namespaces "BlueBox" and "GreenBox" to deploy container with different service version. In a routing rule of ISTIO, we predefined two destination environment for BlueBox and GreenBox, and control the request traffic routing with managing the workload weight in the destination rules.

In order to reduce the impact of noise data in our experiment environments, we run the experiments ten times for each use case in each environment, and used the arithmetic mean to get the average performance metrics as detailed in the next section.

4.2 Blue/Green Deployment Performance Metrics

Ideally, the switch from Blue to Green should be effective immediately, as shown in Fig. 3(A), i.e., when the switch is activated, all incoming traffic requests are immediately routed to the new service release (version 2) without delay or error. However, in reality, the switch is always followed by a period of inconsistency as shown in Fig. 3 (B) when some incoming traffic requests are routed to the current service release (version 1) while other incoming traffic requests are routed to version 2. The reason for this inconsistency dwells within the distributed nature of the information identifying a service instance deployed in the environment. The propagation of the switch from the Blue service version to the Green service version is specific to each implementation of the B/G deployment techniques. To compare the performance of each B/G techniques, we define four analysis metrics, illustrated in Fig. 3(C) and described here after.

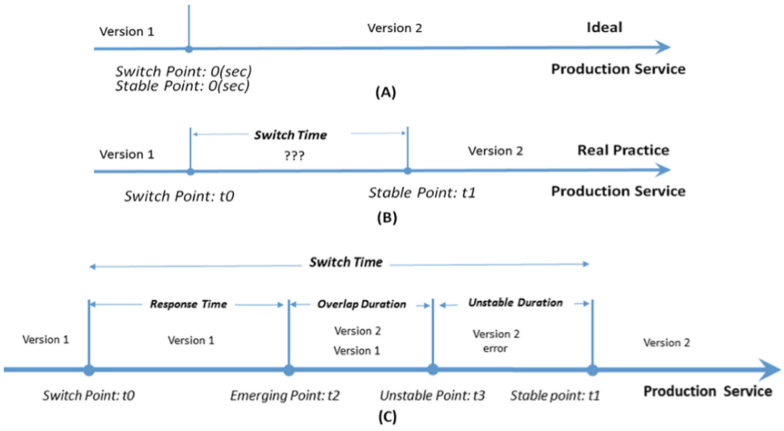

Fig. 3. Comparison between Ideal (A) and real (B) lifecycle of Blue/Green deployment traffic switch; and (C) performance indexes definition for switching Traffic. (Color figure online)

Let t0, called the Switch Point, be the activation time of switching the service versions, i.e., the moment when the information related to the new service version, is made available in the production environment. Let t1, called the Stable Point, be the time corresponding to a consistent successful response to the incoming traffic from the new service version without any response from the original service version. The time difference between t1 and t0, called the Switch Time as Eq. (1), indicates the duration of the B/G deployment and it is our first comparison metric.

$$Switch\,Time = t1 - t0 \tag{1}$$

In the ideal case, the *Switch Time* is zero given that no delay or error occurred when switching the releases. In the real case, we aim to minimize this value.

Let t2, called the *Emerging Point*, be the time when the new service version is observed for the first time in reply to the incoming traffic. The time difference between t2 and t0 indicates the *Response Time* as Eq. (2), to the activation of the switch to the new service version. This is the second comparison metric.

$$Response\,Time = t2 - t0 \tag{2}$$

Let t3, called the *Unstable Point*, indicate the time when the new service version becomes unavailable resulting in an error reply to the incoming traffic. We call the *Unstable Duration* as Eq. (3), the time interval when the new service version is unavailable during the switch period. This is our third comparison metric.

$$Unstable\,Duration = t1 - t3 \tag{3}$$

Finally, our forth comparison metric is the *Overlap Duration* as Eq. (4) which indicates the time interval when the service's two versions are both observed in reply to the incoming traffic. This is the error free time between the *Emerging Point t2* and the *Stable Point t1*, as follows:

$$Overlap\,Duration = t1 - t2 - Unstable\,Duration \tag{4}$$

The aim of all the B/G deployment techniques is to minimize the *Switch Time*:

$$Min\,(Switch\,Time) = Min\,(Response\,Time,\ Overlap\,Duration,\ Unstable\,Duration) \tag{5}$$

5 Experimental Results and Analysis

In the experiments for AWS R53 DNS based B/G deployment, we observed that the Response Time is about 3 min (178 s), while the *Overlap Duration* is about 10 s, as shown in Fig. 4. The root cause for such a large delay on the *Response Time* is due to the DNS functionality, i.e., its caching mechanism used to speeds up the process by storing information for periods of time and re-using it for future DNS queries.

Besides the cache on the nodes in the routing path towards the target service instance, the client and the browser could also use local cache for the target domain name. Thus, there is a long period to update all cache systems on the routing path when the DNS configuration is updated to point to another service instance (i.e., version 2).

Additionally, if the original service instance is still up, the requests will observe an overlap of service response as version 1 or version 2 randomly on different routing path due to the cache.

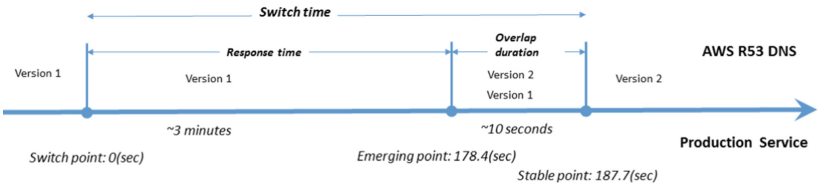

Fig. 4. Experiment result & analysis for AWS R53 DNS based Blue/Green Deployment. (Color figure online)

The experimental results for AWS LB-ASG based B/G deployment are illustrated in Fig. 5 and show a shorter *Response Time* than those in the previous DNS based solution. This is due to the requests being routed to the same endpoint of the Load Balancer (LB). The LB is configured with the internal routing rule to forward the requests to a working node in the target group which is integrated with ASG [11]. However, there is delay for the scale-in/scale-out nodes in AWS ASG, which causes an

overlap when both version 1 node and version 2 node are present at the same time in the group. This confuses the LB into sending requests to version 1 node which impacts the *Overlap Duration* in this B/G deployment. Moreover, even when the version 1 node is removed from ASG, the LB could still send request to the removed service instance due to health status update delay, which will lead to a response error (when the service is unavailable). Therefore we observe an *Unstable Duration* in these experiments.

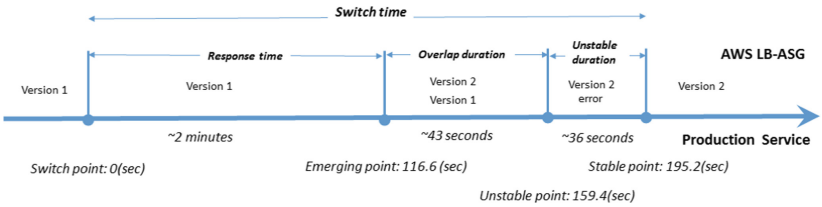

Fig. 5. Experiment result & analysis for AWS LB-ASG based Blue/Green Deployment. (Color figure online)

In the experiments for CF-RR based B/G deployment, the *Switch Time* is smaller than in the previous experiments (DNS based and LB-ASG based), as shown in Fig. 6. The *Response Time* is only about 18 s when another version (version 2) in Green environment is emerging in the request responses. As in the previous solutions, a version overlap is again observed ($\sim$7 s) when we switch the traffic from version 1 to version 2. After analyzing this technique [12], we found the root cause being the sequence of the CF CLI execution for mapping and unmapping the route between the Blue and Green service instances. The CLI execution takes time to make the mapping/unmapping operational. If we change the sequence of the CF CLI execution, to execute first "unmapping blue.example.com from blue", and then "mapping green to blue.example.com", we could remove the overlap. However, the risks is to render the service unavailable when the route of blue.example.com would be requested without an instance mapping.

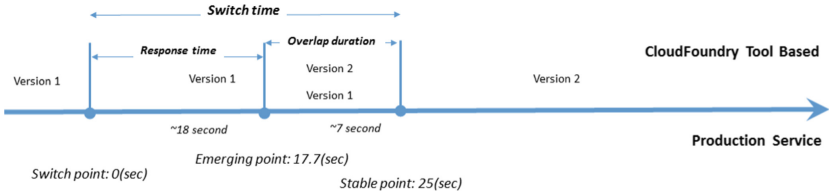

Fig. 6. Experiment result & analysis for CF route remapping based Blue/Green Deployment

In the experiments for our Eureka and Zuul based B/G deployment, we got similar performance results on the Switch Time with those in the CF-RR based B/G deployment. Additionally, no overlap or unstable duration was observed, as illustrated in Fig. 7. It is the service discovery direct configurations and its cache update mechanism which are different from the methods used in the AWS EC2 services and CF tools. In the Eureka and Zuul based B/G deployment implementation, there are multiple configurations we can customize.

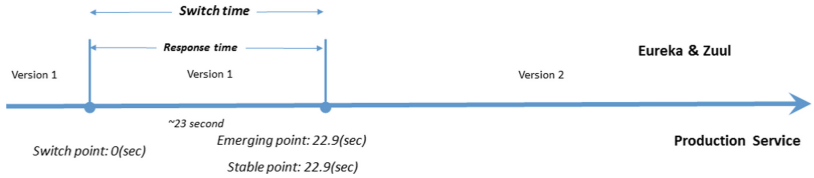

Fig. 7. Experiment result & analysis for Eureka and Zuul based Blue/Green Deployment. (Color figure online)

As presented in Sect. 4, we optimized the time for service registry and cache update. Those optimizations are used to make sure the service instances in the Blue and Green environments register and de-register from the service registration and discovery server (Eureka) synchronously to minimize the switch time, while informing the router (Zuul) of those registry service instances expediently. To minimize the *Switch Time*, besides minimizing the *Response Time* (i.e., discover registry service update in time), we are also trying to minimize the *Overlap Duration* and *Unstable Duration*. This is achieved by controlling the cache content and the cache update interval for the Zuul server, since Zuul manages all the requests routing to the backend service instances. Given that we keep only one service instance (Blue or Green) in the service registry server, the ambiguity for the response (the service version in our experiments) is eliminated in this technique. Additionally, since the service instances are kept running in both the Blue and the Green environments, the risk for service unavailability is avoided even when we keep one service instance in the service cache of Zuul.

The experiments for Kube and ISTIO based B/G deployment, resulted in the best performance results on the *Switch Time* comparing with other B/G deployment solutions, with less than 1 s for switching as illustrated in Fig. 8. The traffic shifting is achieved with updating ISTIO routing policy as described in Sect. 4. Similar to the Netflix Eureka and Zuul solution, the Kube and ISTIO based B/G deployment shows no overlap or unstable duration. The traffic switch takes place immediately within 1 s which only causes a longer response time for the requests in transaction with from new version.

Fig. 8. Experiment result & analysis for KUBE and ISTIO based Blue/Green deployment. (Color figure online)

Table 1 summarizes the performance metrics values of our comparison between the five solutions of B/G deployment. It shows the average results of the performance across 10 sets of experiments for each solution, and the standard deviation values for those experiments in each solution.

Table 1. Blue/Green switch performance comparison summary

Metric (sec)	Switch traffic solutions				
	AWS R53-DNS based	AWS LB-ASG based	CF RR based	Zuul & Eureka	Kube & ISTIO
Response time	178.4	116.6	17.7	22.9	0.087
Standard deviation of RT	36.6	9.8	4.8	5.5	0.025
Overlap duration (OD)	9.3	42.8	7.3	0	0
Standard deviation of RT	6.1	8.6	6.3	0	0
Unstable duration (UD)	–	35.8	–	–	–
Standard Deviation of UD	–	9.8	–	–	–
Switch time (ST)	187.7	195.2	25	22.9	0.087
Standard deviation of ST	32.3	18.5	3.8	5.5	0.025

The DNS based solution which is a simple and general solution for switching traffic to a new service instance, takes the longest time to switch the traffic due to its usage of cache, and introduces service version response ambiguity on the application/client side. Moreover, this method exhibits the maximum standard deviation for its metrics, which means its performance is the most unstable.

For the LB-ASG based solution, it is the service instance initialization from the Launch Template which takes very long during the response time interval. The performance could be improved if the Launch Template can support containers. Moreover, the AMI images creation for each service instance in the template is an extra load for the B/G deployment, which also limits the agility of the new version release.

The Service Discovery based method shows the best performance for switching traffic due to is minimal *Overlap Duration* time cost. This approach also solved the issue of the service unavailability, although it requires customized configurations and sidecar installation on the nodes of the B/G environments, which is an extra load that we addressed via automation pipeline (e.g. UCD). All service instances are running via Kube DNS while ISTIO provides traffic routing in the service mesh for our container environment on the Kube special "VPN" (virtual private network) without outside network routing path. This is key to eliminate the cache on the routing path shown in other solutions and which impacts the overlap duration performance.

6 Conclusion

In this paper, we discussed a continuous delivery methodology, Blue/Green deployment. The most prevalent solutions for implementing Blue/Green deployments were investigated and their performance compared and analyzed. The DNS based solution provides a simple approach that can be used in environments equipped with DNS servers. However, it performs very poor when it comes to switching over traffic between service releases. AWS Load Balancer & Auto-Scaling-Group (ASG) based solution can achieve cost efficiency continuous delivery by keeping only one environment running. However, it takes a relatively long time to initialize a new service instance update. CloudFoundry Remapping Router (CF-RR) is an approach to update a service's route mapping which showed a good response time. CF-RR switches traffic from a service's old version to the new version once the new version becomes available. However, an overlap was observed when using this approach because there is a delay for client commands to take effect. CF-RR and AGS share the same weakness, this is they both work only for services running in their respective platforms. Lastly, the Service Discovery solution exhibited a better overall switch over performance by removing the overlap and unstable periods. The Eureka and Zuul solution however spends more time on the response phase than the CF-RR solution. Kube and ISTIO solution shows the best performance for the switch time, with the caveat that it only works for Kubernetes based environments. Eureka and Zuul approach provides a general way to support any services.

Based on the analysis of the characteristics of each solution, it is important to choose a suitable Blue/Green deployment for service continuous delivery according to the run-time conditions. Getting services up and running quickly while achieving upgradeability and easy to manage deployments with minimized risk, are key to delivering fast and reliable deployments of new technology investments.

Our future work will focus on investigating the more complex scenario of upgrade on hybrid cloud. In such case, the challenge will be the cache synchronization among nodes on different clouds with different service discovery instances, while minimizing the overlap duration and service instance conflict.

References

1. Humble, J., Farley, D.: Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation (Adobe Reader). Pearson Education, London (2010)
2. Chen, L.: Continuous delivery: huge benefits, but challenges too. IEEE Softw. **32**(2), 50–54 (2015)
3. Soni, M.: End to end automation on cloud with build pipeline: the case for DevOps in insurance industry, continuous integration, continuous testing, and continuous delivery. In: IEEE Cloud Computing in Emerging Markets (CCEM), pp. 85–89, 25 November 2015
4. Fowler, M.: Blue Green Deployment (2010). https://martinfowler.com/bliki/BlueGreenDeployment.html
5. https://searchitoperations.techtarget.com/definition/blue-green-deployment
6. https://medium.com/netflix-techblog/deploying-the-netflix-api-79b6176cc3f0
7. Cloudflare global managed DNS: https://www.cloudflare.com/dns
8. How to set up a host name with DigitalOcean. https://www.digitalocean.com/community/tutorials/how-to-set-up-a-host-name-with-digitalocean
9. Google Cloud DNS. https://cloud.google.com/dns/docs/
10. Ellis, N.: An example Blue/Green deployment using Tutum and Cloudflare (for DNS) (2016). https://gist.github.com/neilellis/2d25f0ade3d6cae6f7c9
11. Amazon: Blue/Green deployments on AWS. Whitepaper, August 2016. https://d0.awsstatic.com/whitepapers/AWS_Blue_Green_Deployments.pdf
12. Cloud Foundry: Using Blue-Green deployment to reduce downtime and risk. https://docs.cloudfoundry.org/devguide/deploy-apps/Blue/Green.html#map-green
13. Digital Ocean: How to use Blue-Green deployments to release software safely. https://www.digitalocean.com/community/tutorials/how-to-use-Blue/Green-deployments-to-release-software-safely
14. Danial S.: Thought Works, Implementing Blue-Green deployments with AWS (2013) https://www.thoughtworks.com/insights/blog/implementing-Blue/Green-deployments-aws
15. IBM UrbanCode Deploy. https://developer.ibm.com/urbancode/products/urbancode-deploy/
16. Klusak, V.: Klokan Technologies, Blue-Green Deployment with Docker and Nginx (2016). https://blog.klokantech.com/2016/08/Blue/Green-deployment-with-docker-and.html
17. Pérez, I.S.: Simple Blue/Green deployments with Docker and Nginx (2016). http://dukebody.com/?p=511
18. Holý, J.: DZone/Devops Zone, WebApp Blue/Green Deployment Without Breaking Sessions (2016). https://dzone.com/articles/webapp-bluegreen-deployment
19. Netflix Eureka. https://github.com/Netflix/eureka/wiki
20. Netflix Zuul. https://github.com/Netflix/zuul/wiki
21. Zero-Downtime Rolling Deployments With Netflix's Eureka and Zuul, March 2019. https://www.credera.com/blog/technology-solutions/zero-downtime-rolling-deployments-netflixs-eureka-zuul/
22. Janakiram, M.S.V.: Blue/Green Deployments with Kubernetes and Istio, October 2018https://thenewstack.io/tutorial-blue-green-deployments-with-kubernetes-and-istio/
23. Istio. https://istio.io
24. IBM Cloud Toolchain. https://cloud.ibm.com/devops/create?bss_account=49f48a067ac4433a911740653049e83d&ims_account=167466