IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 5 May 2024, accepted 10 June 2024, date of publication 18 June 2024, date of current version 28 June 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3416087

METHODS

Evaluate Canary Deployment Techniques Using Kubernetes, Istio, and Liquibase for Cloud Native Enterprise Applications to Achieve Zero Downtime for Continuous Deployments

ANTRA MALHOTRA[1], (Senior Member, IEEE), AMR ELSAYED[2], RANDOLPH TORRES[3], AND SRINIVAS VENKATRAMAN[4]

[1]System Engineering, Network Systems–Shared Platforms and Product, Verizon, Temple Terrace, FL 33637, USA
[2]System Architecture, Network Systems–Shared Platforms and Product, Verizon, Temple Terrace, FL 33637, USA
[3]Technical Strategy, Network Systems–Shared Platforms and Product, Verizon, Temple Terrace, FL 33637, USA
[4]System Engineering, Network Systems–Shared Platforms and Product, Verizon, Basking Ridge, NJ 07920, USA

Corresponding author: Antra Malhotra (antra.malhotra@verizon.com)

This work was supported by the Shared Platforms and Product Organization, part of Network Systems in Verizon.

ABSTRACT To cater to the changing needs of the businesses, enterprises are adopting processes that allow rapid iteration and feedback loop. Today, development teams work closely with the business leveraging agile methods to gather feedback, assess the impact of the changes and deploy changes in a short duration. By taking advantage of the microservices architecture (MSA), large monolithic code is logically broken down into microservices that can be developed, deployed and scaled independently. Applications are leveraging containerization and orchestration technologies along with microservices architecture to package, deploy and manage the code across different environments. The underlying infrastructure and agile processes need to be supported with robust methods to perform code integration and deployments without any service disruption. This paper provides a qualitative assessment of the following code deployment techniques (i) Recreating Deployments (ii) Rolling Deployment (iii) Blue-Green Deployment (iv) Canary Deployment. This assessment can guide enterprises to identify the right code deployment strategy that can be adopted based on the business use case. Next, the paper dives deeper on how the in-built capabilities of Kubernetes along with open-source tools like Istio can be leveraged successfully to implement canary deployments for service changes. The paper presents a novel technique for performing canary deployments whereby service and database changes can be promoted to production by leveraging Istio and Liquibase along with load balancer without incurring any downtime for the application. The paper provides a complete canary deployment reference architecture that can be adopted by enterprises pursuing zero downtime for continuous deployments.

INDEX TERMS Zero downtime code deployments, canary deployment, high availability, cloud computing.

I. INTRODUCTION

Zero downtime deployment means that a new code or bug fix is deployed and made available to the users without incurring any application downtime or service interruption. The deployments are seamless for the user and their experience is maintained throughout the update process as explained in [1]. Such a deployment strategy enables more frequent changes to be deployed. Modern applications are anchored on the microservices architecture where the application is

The associate editor coordinating the review of this manuscript and approving it for publication was Christian Pilato.

© 2024 The Authors. This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License.
For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

made of independent, loosely coupled services. This modular design helps with faster development cycles, but it also increases the complexity of the application. Given the ever-changing needs of the business, enterprise organizations are looking for opportunities to simplify and optimize the software delivery lifecycle. There is a growing demand to adopt the right techniques to successfully integrate, test and deploy the code without any service disruption. Hence, the paper provides a qualitative assessment of the different code deployment methods to help the enterprise organizations to adopt the right strategy based on the business use case and requirement. In addition, the paper evaluates the canary deployment technique for service changes using Kubernetes gateway API (Application Programming Interface) and Istio service mesh. The paper does a deep dive on canary deployment and proposes a novel technique to rollout changes both for services and database without incurring any application downtime. The technique explained in this paper can also be applied for rolling back changes if not certified. Organizations struggle to maintain zero downtime for database changes and to mitigate this challenge, the paper provides a complete reference architecture that organizations can adopt to achieve true zero downtime using canary deployment technique.

A. CONTINUOUS INTEGRATION AND CONTINUOUS DEPLOYMENT

Continuous Integration and Continuous Deployment (CI & CD) are important practices implemented as part of the software development lifecycle to improve the software delivery process. The workflow shared in the Fig. 1 below shows the steps involved for performing continuous integration and deployment. For integration, code need to be checked into a common repository and then using the automation the new code is merged with the existing code. As part of integration, regression and progression test cases are executed for quality assurance. This process promotes a high level of accountability amongst the developers as the code is scanned for any quality issues, vulnerabilities and functional testing is performed prior to the code merge. While continuous integration focuses on automating the build and test processes to ensure quality and consistency in the code, continuous deployment is focused on automating the process of deploying code changes to test, staging and production environments. The process for continuous integration and deployment is also explained in [2] and [3].

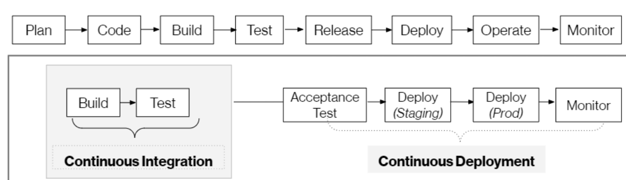

FIGURE 1. Process flow of a continuous integration and continuous deployment pipeline.

Now, in the absence of the continuous integration and continuous deployment practices, the code changes are manually promoted in the test environment for validation. The changes are manually version controlled using systems like Git. The code is built using tools like Maven, Gradle and then compiled in a Java ARchive (JAR) or Web Application Archive (WAR) file. During the deployment, production instances are stopped and at times backed up to serve as a rollback. These activities incur a downtime and the existing code files are replaced with the new build. This approach is manual, linear, slow and requires a downtime for the application. This may work for small, less complex projects and when the business is able to manage the downtime and longer release cycles. However, that is no longer the case with modern enterprises as they are looking for zero downtime for code deployments and zero failure rate. The survey of the various practices conducted by S. Bobrovskis and A. Jurenoks in [4] suggests that modern businesses and software development practices are striving for zero downtime. Zero downtime for code deployments can be calculated as the time difference between when the code is committed to the time the code is available in production. Similarly, zero failure rate can be calculated as the number of failure transactions out of the total transactions processed. The experience created by errors in production post releases and its impact on user experience is well explained in a roundtable discussion mentioned in [5]. It is ideal to keep the difference to zero and also the failure rate to zero when organizations are working towards zero downtime experience for the business.

There is ongoing research on various code deployment strategies and methods to achieve zero downtime. Big giants like Facebook, Netflix have been constantly upscaling their DevOps strategy for zero downtime as explained in [6]. There has been research done on comparing the various techniques to perform Blue/Green deployments. C.K. Rudrabhatla in the paper [7] explores the different techniques to perform zero downtime deployments and then simulates blue/green deployment for a research project using DNS swap, load balancer and newer image switch techniques. Similarly, in another paper [8] Bo Yang and other authors have explored the opportunity to perform service discovery based blue/green deployments for better performance. Jez Humble and Dave Farley in their book on Continuous Delivery [9] introduced the concept of achieving zero downtime with canary deployments. Alexander Tarvo and other authors in their paper [10] elaborated on canary testing and introduced this tool called Canary Advisor to monitor the deployed versions of an application and detect any degradation for performance and scalability. Ernst and other authors in their paper [11] proposed an approach to actively control the request distribution to shorten the time a canary runs for collecting the results. Nichil Stresser compares the different canary deployment tools in the master's thesis [12]. Similarly, in another paper [13] the authors have explored the concept of multi-cloud deployments using Kubernetes with Istio and monitoring tools like Prometheus and Grafana. Given all

the work done, it is ascertained that deployment techniques hold a strong relevance for cloud native applications striving towards zero downtime.

In this paper the focus is on providing a complete reference architecture for canary deployments. This reference architecture provides a novel technique for performing canary deployments using Istio, Liquibase and load balancer to show how both services and database changes can be deployed and rolled back without incurring any downtime.

B. KEY CONTRIBUTIONS OF THE PAPER

The paper introduces novel techniques to overcome the technical challenges associated with system unavailability during code deployments and database upgrades or changes.

1) MAINTAINING ZERO DOWNTIME FOR MICROSERVICE CHANGES

An application has to regularly stay updated and manage service changes for rolling out new features, enhancement or for applying security patches. Maintaining downtime for service changes and upgrades is important for seamless operations and customer experience. The paper explains the different deployment strategies that can be adopted by the organizations for managing the application level changes. The paper does a deep dive on canary deployments and shows how leveraging service mesh and Kubernetes gateway API, the traffic can be redistributed from the old version of the application to the new version of the application without incurring any downtime. In a canary deployment, a new service version is exposed to an early sub-segment of the users with the objective to certify the new functionality prior to making it available for the entire user base. The paper acknowledges that the challenge with maintaining zero downtime with the deployment strategies discussed is that while these strategies can be applied for microservice changes, these strategies are not successful in maintaining downtime for database updates. In case of microservices there can be multiple versions of the microservices (old and new) running to ensure zero downtime, however to manage database schema changes whilst keeping the multiple microservices of the applications running is not feasible.There is no support offered by Kubernetes to make the old and new version of the database schema available along with the application. The paper discusses the novel technique that overcomes the technical challenges associated with maintaining zero downtime for database changes.

2) MAINTAINING ZERO DOWNTIME FOR DATABASE CHANGES

An application uses data that is stored in databases maintained within a cloud computing environment. Certain applications may read data from the database, while other applications may write data to the database. When the database is being upgraded, the existing database that utilizes the database are taken offline. Once an upgraded version of the database is deployed within the cloud computing environment, the applications are provided access to the upgraded database. This results in downtime for the application and its users. The paper explains a progressive rollout strategy that suggests that during database upgrade, incoming requests can be routed to the new version of the database and some percentage of the requests can be routed to the old version of the database. The requests are gradually moved to the new version of the database and the modifications to the different versions of the database are then synchronized with the use of triggers that are used to propagate changes between the different versions of the database in order to keep the database versions in-sync.

3) MAINTAINING ZERO DOWNTIME FOR BOTH MICROSERVICE AND DATABASE CHANGES

The technique proposed in this paper advances the approach and technique of canary deployments by having both old and new versions of the application along with their schema versions to coexist with each other. The pattern used and tested here can support data-intensive applications for high availability without replicating the database. Similarly, the technique supports single or multiple databases as well as single or multiple schemas or even multiple databases with multiple schemas. The solution presented in this paper involves combining Istio which is a service mesh with Liquibase which is a database schema version control tool along with load balancer and traffic router to rollout the changes to a subset of users or servers without incurring any downtime.

Conventional upgrade techniques take an application and database offline during upgrade. This leads to unacceptable downtime for the application users. The upgraded application and database are not available for use until the deployment is complete and fully validated. The technique explained in this paper will enable the application and database upgrades to be completed and validated without incurring any downtime.

In the following Section II, the paper explains the history of continuous integration and deployment process and how the software development and delivery landscape has evolved to meet the changing needs of the business. Section III covers the various building blocks that enterprise organizations can adopt for implementing continuous deployment techniques. Section IV provides a qualitative evaluation of the different continuous deployment methods (i) Recreating Deployments (Purge) (ii) Rolling Deployment (iii) Blue-Green Deployment (iv) Canary Deployment. The qualitative evaluation helps to assess the right code deployment strategy that can be adopted for achieving zero downtime. Section V presents a deep dive on canary deployment by explaining the canary deployment workflow and introduces the Kubernetes gateway API and open-source technologies that can be leveraged for managing application-level changes. In Section VI, the paper does a comparison between two techniques to perform canary deployment and evaluates the most efficient way to perform canary deployment for application changes. The two techniques discussed in this section perform canary deployment with Kubernetes gateway API and service mesh and another technique that involves use of service mesh only.

In Section VII, the paper expands the techniques compared in Section VI to also include managing database changes without incurring any downtime. This section provides a complete reference architecture that can be leveraged by enterprise organizations for incorporating canary deployments for managing zero downtime for both application and database changes. The last Section VIII, gives a summary of the paper and draws inferences based on the techniques discussed.

II. HISTORY AND RELATED WORKS

The early days of software development were associated with small and isolated teams. Code was written without much standardized practices and the software development life cycle followed a linear sequence where the phases of requirements, design, development, testing, implementation and maintenance were managed sequentially. Developers worked in silos and code was merged from different teams leading to integration challenges. This phase was characterized by manual deployment and lack of automation. In 1991, Grady Booch advocated frequent use of classes and objects in programming to simplify software design and this concept of object that enabled abstraction for complex software systems is further explained in [14]. Grady's intent was more on simplifying the design rather than promoting frequent software changes. However, in 1997 Extreme Programming was introduced and it recognized the need to improve software quality and responsiveness to the changing business needs. Extreme Programming introduced peer programming, shorter release cycle, code reviews, unit testing and user acceptance testing as explained in [15]. Eventually, these steps formed an integral part of the software development lifecycle. Other methodologies like Scrum, Kanban, were introduced with a common goal to build better quality software and release it to the business within a shorter time frame. The effects of these methodologies on software development are further explained in [16] and these methods played a big role in establishing various agile frameworks for software delivery. While the software industry recognized that it was important to deliver software more quickly, there was a lack of adequate tools and automation in place to do so. In 2001, CruiseControl was introduced and this was a game changer as it automated builds, testing and commits to a version control system. This enabled continuous integration and also helped in identifying any integration conflicts early in the process. During the late 2000's, the concept of continuous deployment started to take shape with the introduction of tools like Jenkins. Jenkins not only enabled automated testing but also automated deployment to staging environments. Configuration management and ability to consistently and repeatedly deploy to environments was made possible with the help of tools like Puppet and Chef. In the mid-2010's the new applications were being built using the microservices architecture, where services could be independently developed, tested and deployed. Next came the rise of cloud computing, containerization technologies like Docker and container orchestration platforms like Kubernetes. These technologies introduced self-managed, self-hosted code integration and deployment tools that are able to operate with cloud environments and can be integrated with services for observability and analytics. The Fig. 2 below shows a complete timeline of evolution of continuous integration and continuous deployment techniques leading to modern day containerization. The history of continuous integration, deployment and overall software engineering is further discussed in [17] and [18]. The latest development with microservices architecture and containerization is oriented towards keeping the application downtime minimum, detect issues early on in the process and leverage automation and version control practices.

1990's	2000's	2010's
• Classes • Object Oriented Programming • Extreme Programming	• Cruise Control • Scrum • Kanban • Jenkins • **Continuous Integration** • **Continuous Deployment**	• Cloud Technologies • Microservices Architecture • Containerization

FIGURE 2. Timeline of evolution of continuous integration and continuous deployment leading to modern day containerization.

In this paper we will review the code deployment methods to understand the pros and cons associated with these methods. These techniques will be qualitatively evaluated based on performance, user experience and cost. The evaluation will help organizations to understand the right technique that can be adopted based on the business use case. Even though, from a technology perspective we have come a long way in managing continuous integration and deployment. However, keeping zero downtime is still a challenge when there are application and database schema changes involved. Most of the organizations are challenged to perform database upgrades and maintain zero downtime as these changes could be sensitive, time consuming and can compromise the data integrity if not done properly. The paper provides a novel technique to perform canary deployment using Istio, Liquibase and load balancer for both service and database changes. Using this technique, applications can maintain zero downtime during code deployments. The paper presents a complete reference architecture for canary deployments that can be adopted to maintain zero downtime for both services and database changes or upgrades.

III. BUILDING BLOCKS FOR CONTINUOUS DEPLOYMENT

This section of the paper elaborates on the building blocks for implementing continuous deployment techniques. These best practices can help enterprise organizations build a robust deployment pipeline to deliver the software faster, frequently and without any service interruption. These building blocks are important for achieving zero downtime architecture and are leveraged by the canary deployment technique discussed in this paper for service and database changes.

A. MODULAR DESIGN

The modular design of the application based on microservices architecture enables the application to run and manage the services as independent components that are loosely coupled

together. Each service can be managed, deployed, updated and scaled independently. This transition from monolithic to microservices architecture and emergence of modularity in software design is explained in [19], [20], and [21]. One service does not share any code or implementation with another service. In microservices architecture as provided in Fig. 3 below, the monolithic pattern of writing the code is broken down into smaller chunks for easy maintainability. Prior to microservices, when the applications were written as one big piece of monolithic code and multiple teams worked on the same piece of code base, it was extremely difficult to merge code. Every time a new change had to be deployed, all the changes needed to be queued, integration conflicts resolved, functional testing done for the entire application. Finally, the entire application had to be released as all the new and old code was bundled together as one unit. Thus, in a non-microservices architecture, release management is challenging and requires a lot of coordination across teams.

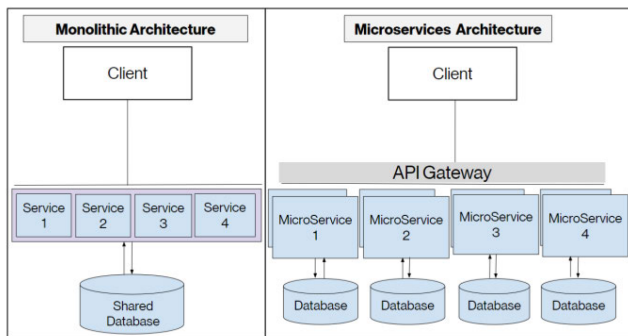

FIGURE 3. **A comparative view between monolithic and microservices based architecture.**

B. CONTAINERIZATION

In a traditional deployment scenario, the operating system is running on hardware and the applications or software runs on top of the operating system using the common libraries and dependencies. When an upgrade or a change is needed for the application, there are conflicts due to the intertwined libraries and dependencies. However, in a containerized environment, a 'container engine' helps to package the libraries and dependencies as one container. These containers provide a runtime environment where developers can build, test and execute the code bundled in one package. This runtime environment includes the application with all its dependencies, libraries, binary files and other configuration files needed to run the application. 'Docker' is an open-source container engine platform that manages containers. Each container sits on top of a container engine (Docker), which in turn sits on top of an operating system. The Fig. 4 below shows the transition of the infrastructure architecture from traditional deployments to containerized deployments and it is further explained in the references [22], [23], [24], [25], [26], [27]. In case of a modern application, containers allow you to independently deploy services into separate containers making the maintenance process more efficient. The containers can be easily moved from one machine to another and hence they can help with easy deployment of applications to multiple environments.

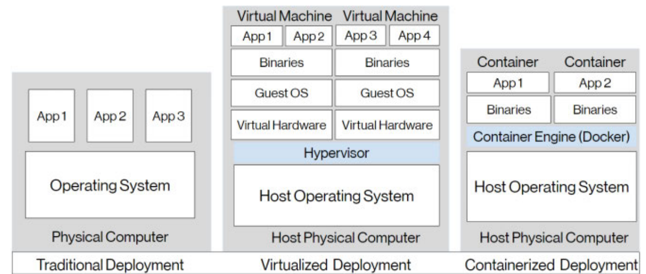

FIGURE 4. **A comparative view between traditional, virtualized and containerized deployment.**

C. GRACEFUL SHUTDOWN

Graceful shutdown requires that all in-flight tasks are completed, resources released properly before the system shuts down. Usually, an application is handling multiple parallel requests from concurrent users and hence it is important to have a certain method to address these inflight requests to prevent data loss and service disruption. Once the server shutdown is initiated, any new incoming requests are rejected and the server shuts down once the last pre-existing request is completed.

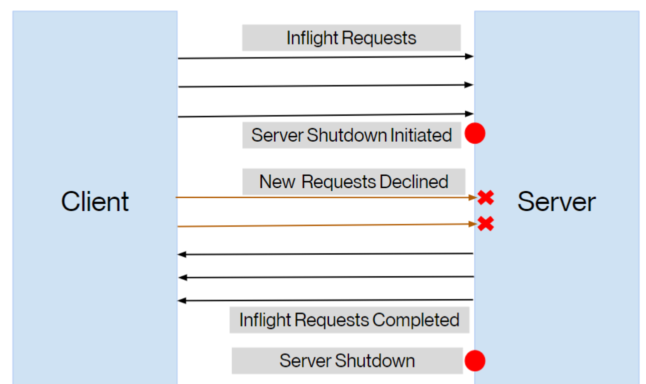

FIGURE 5. **A client service communication showing the handling of incoming and new requests when the shutdown is initiated.**

The Fig. 5 shows the process of shutdown between the client and server and how the requests are handled when a shutdown is initiated. In case of continuous deployment, having a graceful shutdown can ensure a smooth transition from one version of the application to another [28], [29], [30], [31]. Data integrity of the inflight transactions is maintained by gracefully shutting down the application or else if in-flight transactions are lost or partial requests are completed this will lead to service disruption and poor user experience.

D. LOAD BALANCER

A load balancer is responsible for distributing the application traffic across identical servers. In a continuous deployment environment, a load balancer can help move the incoming requests to the server with the old version while the new version is getting deployed on another server. Similarly, if there are issues encountered with the new version, load balancer can move the traffic back to the old version. Load balancers can also help with splitting the traffic to specific versions of the application and this helps to stabilize the new version of the software before making it available for general use. Load balancing is also important for scalability as based on the volume of the incoming traffic the new servers can be added or removed from the load balancer to manage traffic. Load balancing is an integral part of cloud computing and there are many techniques introduced to manage load in a cloud environment as explained in [32] and [33] to take advantage of economies of scale. This overall helps with reliability and scalability of the application. The Fig. 6 below shows how the load balancer is managing the traffic across the servers.

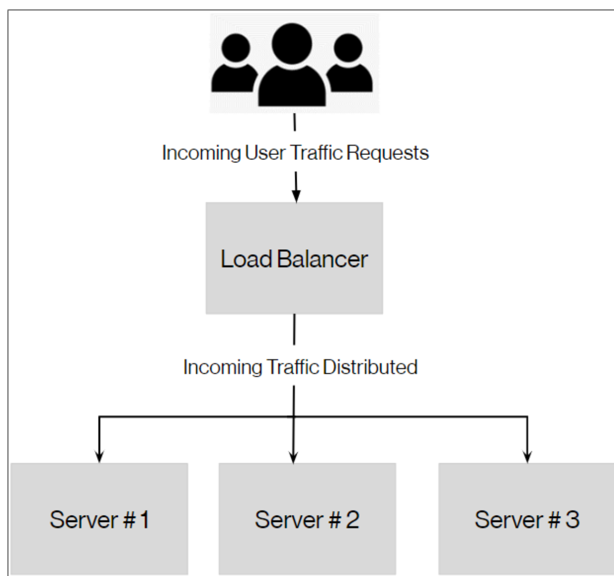

FIGURE 6. A load balancer managing the incoming traffic amongst various application servers.

E. HEALTH CHECKS AND TRAFFIC ROUTING

Health checks and routing traffic dynamically are an extension of the load balancing. When the load balancer is distributing requests to the servers, it can make informed decisions on routing the traffic based on the health checks performed for the servers. This ensures that traffic is always diverted to the healthy servers and there is no service disruption. Load balancing is essential for efficient operation in distributed environments [34]. These health checks can be user initiated whereby the load balancer checks for the health before sending the traffic and then if a particular instance of the server is not responding, the load balancer will mark that instance as unavailable and will send the traffic to other available instances. Similarly, load balancer can do such frequent checks on an alternate end point of the server, these checks are not user initiated and then based on the checks the servers can be made unavailable until back up again to receive traffic. Similarly, dynamic routing can help with seamless transition of the traffic from the old to the new version. It supports controlled testing of the new versions of the software by gradually releasing the traffic to the new versions. The health checks and dynamic routing go hand in hand with load balancing. The Fig. 7 below shows the load balancer managing traffic amongst the different instances based on the health checks. This is integral to the success of continuous deployments and supports application availability and reliability.

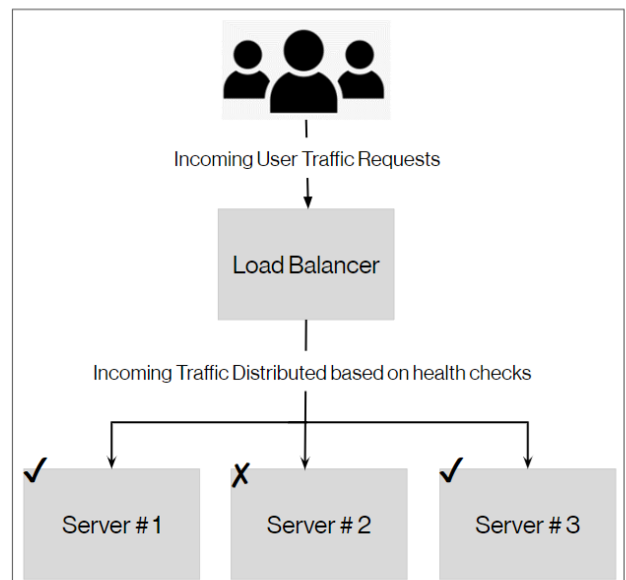

FIGURE 7. A load balancer managing the incoming traffic based on health checks.

The building blocks shared in this section will be leveraged in the solution discussed in this paper for achieving zero downtime during deployment of services and database schema changes.

IV. DEPLOYMENT STRATEGIES

Kubernetes provides a framework for container management and orchestration. It manages the scaling of the pods, failover and deployment patterns as mentioned in [35]. Applications deployed in Kubernetes can take advantage of using the out of box deployment strategies to achieve zero downtime during an upgrade or maintenance window and the [36] by Domingus and Arundel can be referred to understand how the applications can be built, deployed and scaled in a cloud environment. To perform a qualitative analysis on the code deployment techniques, the paper considers a containerized enterprise cloud-native application. This Application (1) has a sample microservice (A) deployed in a Kubernetes

environment and replicated on multiple pods for scalability and high availability.

A. RECREATING DEPLOYMENT (PURGE)

The Application (1) is considering a complete revamp of its microservice (A) version 1 and introducing user interface changes. In this scenario, recreating deployments is used to replace the existing deployment of the service with the new version. This is done by shutting down the existing pods and resources and replacing them with a version 2 of microservice (A) running on a different set of pods. There will be downtime as the old instances are stopped prior to bringing up the new instances. The downtime is a function of the time it takes to shut down the old and then bring back the new instances. Any required configurations will have to be applied to the new service before initiating the new deployment. The new instances should be validated and tested prior to making it available to the entire user base. The Fig. 8 below shows recreating deployments and how the user traffic is moved to the new instances of the application and the old instances are shut down. Such deployments are usually done to manage major releases that involve significant changes to the software or when the application is not capable of handling continuous deployment.

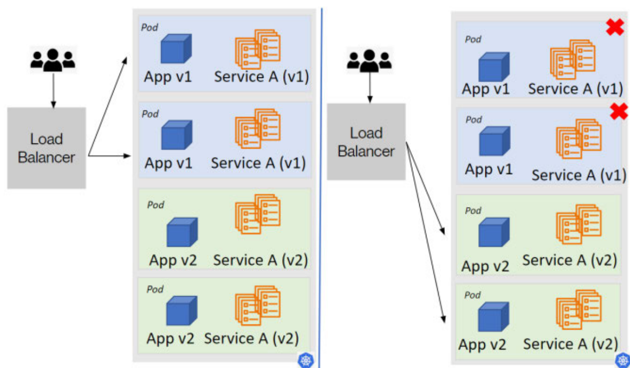

FIGURE 8. Recreating deployment where the traffic is moved 100% to the new Application version and old instances of the pods are shutdown.

B. ROLLING DEPLOYMENT

The Application (1) is making a change to its microservice (A). With rolling deployments, the pods running the older version 1 of the service can be replaced with a newer version 2 while continuing to service the clients. In order to achieve this, a deployment file is created with a configuration about the number of pods to be created at the time of deployment (also known as replica set). Deployment is triggered when the image in the deployment file is updated from version 1 to version 2 for microservice (A). New pods are created pointing to the new image and gradually the old pods are terminated based on the positive health check of the new pods. As the new pods start to come up, traffic is diverted to the new pods. The Fig. 9 shows rolling deployments where the pods running with the old version of the service are replaced with the new version of the service. For the users the transition is seamless as there is no downtime and the traffic gradually moves from the old pods to the new pods and eventually all the old pods are terminated or replaced with the new pods.

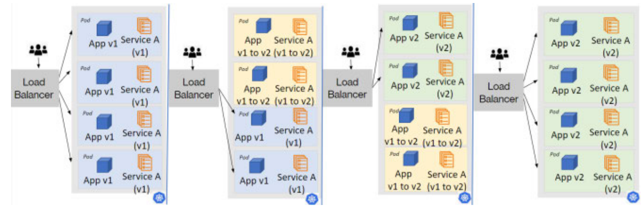

FIGURE 9. Rolling deployment where the pods running with the old version of the service are replaced with the new version while continuing to service the clients.

C. BLUE/GREEN DEPLOYMENT

While the rolling update provides the flexibility to upgrade the service without any service disruption, however the user experience can be poor if the new change that is rolled out is not functionally aligned with the user expectations or it is not able to handle real-life traffic leading to performance degradation. To avoid this scenario, the Application (1) can deploy using the Blue/Green method by creating two identical environments, where 'Blue' is the version 1 of microservice A running in production and 'Green' is the new version 2 available in a parallel production environment but it is not live yet. The traffic from the 'Blue' environment for microservice A version 1 is moved to microservice A version 2 in the 'Green' environment after the new version is fully tested. The 'Blue' version can stay on a standby mode until the new version is stabilized. Blue/Green Deployment requires two exact similar instances of the environment running to support existing and the new version of the code. The traffic can be routed using application load balancers, routers or a reverse proxy. The Fig. 10 shows Blue/Green deployment where the 100% traffic is moved from old to the new version.

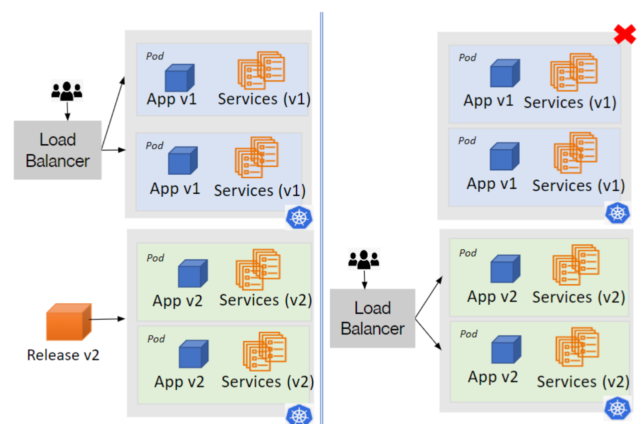

FIGURE 10. 100% traffic is moved to a new instance in green and blue instance can be turned down once the new version is stabilized.

D. CANARY DEPLOYMENT

Canary gives the advantage of making the changes available to a certain set of users for validation and only once certified the new changes can be rolled out for the entire user base. This approach gives the flexibility to validate the changes for performance and stability prior to enabling the new changes to the entire user base. Application (1) has microservice (A) version 1 running in production as a 'stable' version and a new version 2 is introduced. A percentage of users or specific users can be identified as a 'canary group' and the new version of the application can be deployed only for this subset of users. The performance and functionality of the new version can be tested and once stabilized the new version of the microservice (A) can be made available to the entire user base. If issues are detected in the new version, rollback is easy as the entire traffic can be rolled back to the stable version. The Fig. 11 below shows how the initial 20% traffic is sent to the canary version and once validated 100% traffic is moved to the canary version.

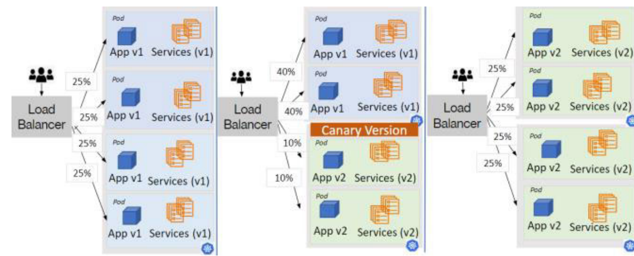

FIGURE 11. 20% traffic is sent to the canary version and once validated 100% traffic is diverted to Canary version post validation.

To summarize, below is the qualitative assessment of the four deployment techniques (i) Recreating Deployment (ii) Rolling Deployment (iii) Blue/Green Deployment (iv) Canary Deployment. The assessment is based on: Lead time for changes, Cost and User Experience. The deployment techniques are assessed as 'High' Medium' and 'Low' against these parameters based on the qualitative assessment Lead time is the function of the time taken to deploy the changes, commit to production and then the time taken to rollback the changes. Cost is assessed based on the resource utilization during the code deployment process and User Experience is the outcome of any service disruption experienced during the deployment. Enterprise organizations striving towards zero downtime deployments should work towards maintaining a low lead time for changes, operational cost associated with the deployment method should be low and the user experience should not be impacted due to any service disruption. The Table 1 below captures the qualitative assessment for the four deployment techniques explained in this section.

The qualitative assessment indicates that canary deployments fare well compared to other techniques and can be a potential tool for enterprises striving towards zero downtime deployments. As there has been limited research done on how canary deployments can be used for achieving zero downtime, the next section of the paper will evaluate canary deployment techniques for managing service changes using Kubernetes gateway API and Istio service mesh. The paper will then extend the technique evaluated to also perform database schema changes along with service changes without incurring any application downtime. The reference architecture provided in this paper for canary deployments can serve as a blueprint for enterprises striving for zero downtime code deployments.

TABLE 1. Qualitative assessment of the deployment techniques.

Criteria	Indicators	Recreating	Rolling	Blue/ Green	Canary
Lead time for changes	Deploy Commit Rollback	High	Medium	Low	Low
Cost	Resource Usage	High	Medium	High	Low
User Experience	Service Disruption	Yes	No	No	No

V. CANARY DEPLOYMENT WORKFLOW AND TECHNIQUES

Canary deployments require a way to manage traffic so the workloads can move successfully from the old version to the canary version. It also requires observability built in to assess the effectiveness of the canary version. The workflow outlined in Fig. 12 below shows the decision-making process throughout the canary deployment.

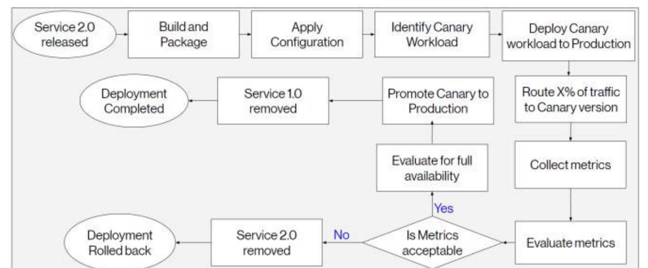

FIGURE 12. Decision making workflow to manage traffic for Canary deployments.

Kubernetes provides in-built features to orchestrate the traffic and also scale the instances based on the incoming traffic. However, for canary deployments that involve application and database changes, the in-built features of Kubernetes are not adequate and should be supported by open-source tools like Liquibase, Istio and load balancer to achieve true zero downtime for the application. CNCF (Cloud Native Computing Foundation) and the CDF (Continuous Delivery Foundation) shelters open-source competitive tools to improve enterprise's ability to deliver software with speed and security. Details about the open-source tools and the process to manage the tool is further explained in [37] and [38]. In this section, the paper elaborates on the Kubernetes services and explains the reason to introduce open-source tools to perform canary deployments.

A. SERVICES

Elaborating on the Kubernetes capabilities to enable zero downtime, a pod is the smallest deployable unit in Kubernetes and it consists of multiple containers running on a cluster. Kubernetes has a set of services that can be used to expose a set of pods running the application instances to the external world or even for internal communication amongst the pods within a cluster. Usually, pods are deployed with multiple replicas of the application instance for scalability and reliability. However, the pod gets a new IP address every time a pod is replaced due to failure or a new one added to manage the workloads. Hence, the Kubernetes services provides the abstraction for a stable IP address and DNS name to facilitate external and pod-to-pod communication. Network policies can be applied to manage traffic to the pods and how the pods communicate to each other. There are different types of services offered by Kubernetes that can be utilized based on the use case. These different types of services are ClusterIP, NodePort and LoadBalancer [39]. ClusterIP enables the service to be reachable by other pods within the cluster by assigning its own IP address. NodePort will expose the service on a static port on each node's IP address. This allows the services to be accessible from outside the cluster. LoadBalancer will create an external IP and then manage traffic to either NodePort or ClusterIP. For the purpose of this assignment, we have used ClusterIP and LoadBalancer service in our setup.

B. LEVERAGING OPEN-SOURCE TECHNOLOGIES

Now, given the understanding of the different deployment strategies and Kubernetes service capabilities, if canary deployment had to be performed where Version A is the stable version and Version B is the canary version then performing canary deployment using Kubernetes native capabilities comes with challenges due to the way Kubernetes cluster works from a traffic routing perspective.

Kubernetes does not provide the capability to perform rolling update deployment with Version B of the service and expect that Version B will remain up and running. This is because the way Kubernetes rolling updates work is that for a single deployment, Version A will have as many replicas scaled and then once Version B replicas are introduced and stabilized, Version B replicas will become primary and Version A replicas will be turned down. In order to perform this exercise, two separate deployments of the same service will have to be performed. Now, if two separate deployments are used for each version then each deployment will have its own auto-scaler which is a good concept when both the deployments need to be kept available. However, this imposes a gap for canary deployment where 90% traffic needs to be diverted to Version A deployment and 10% traffic to Version B deployment. This kind of traffic distribution cannot be achieved as each deployment will have its own auto-scaler running in silo from the other version and it will scale up only based on its load. Thereby giving very little opportunity to control traffic segmentation for each version. Hence, manual scaling will be needed for Version A to be at 9 replicas and Version B to be at 1 replica. Now, this is a tedious process as every time the replica count will have to be readjusted based on the traffic. So, even if a single deployment or two deployments are used, it will not help as there is a fundamental gap with Kubernetes because it uses instance scaling to manage traffic version distribution and replica deployments. Hence, a better solution is required to manage canary deployment and open-source tools like Istio [40], [41] that offer solution for managing scaling of the pods, introduce the use of VirtualService and DestinationRule objects to decouple traffic distribution from deployment replicas by using an application load balancer level controller that does the heavy lifting of traffic splitting without having to deal with replicas ratio challenge. Hence, for the purpose of this paper canary deployment for service changes will be executed using two options. Option 1 involves performing canary deployment using Kubernetes gateway API and Istio service mesh and option 2 involves performing canary deployment using Istio service mesh only. This evaluation will show how Kubernetes framework when augmented with these open-source tools can enable zero downtime deployments for service changes.

VI. EVALUATE CANARY DEPLOYMENT TECHNIQUE

This section will cover the metrics collected when canary deployment is executed for service changes with Option 1 using Kubernetes gateway API and Istio service mesh and then Option 2 using only Istio service mesh. The deployment methods are evaluated based on performance, efficiency and any failures reported. To be able to measure the deployments based on these criteria the parameters identified include Time taken for tests, Requests per second, Time per concurrent requests and Number of failed requests. The Table 2 below captures the formula for the metrics identified for comparison and this is also referenced in [42].

TABLE 2. Metrics for canary deployment test runs.

Metrics	Calculation
Time taken for tests	This is the time taken from the moment the first socket connection is created to the moment the last response is received
Requests per seconds	This is the number of requests per second. This is calculated as *Total number of requests/total time taken for tests*
Time per request	The average time spent per request
Time per concurrent requests	The average time spent per request based on the concurrency level *Concurrency * time taken * 1000/requests done*
Failed requests	Count of Failed Request out of Total requests

A docker desktop enabled with Kubernetes single-node cluster is set up to execute the canary deployment. The Table 3 below elaborates on the namespaces created to apply gateway configurations.

A spring boot containerized microservice 'HelloStats' with basic health check and spring boot actuator HTTP endpoint

TABLE 3. Kubernetes namespaces for the deployment.

Name Space	Usage
zdt-gw-stats-istio	Namespace with istio plus K8S gateway helm configuration
zdt-mesh-stats-istio	Namespace with istio service mesh helm configuration
istio-ingress	Namespace to install istio ingress gateway
istio-system	Namespace to install istio system
kiali-operator	Namespace to install kiali the console for istio service mesh
consul	Namespace to install consul the service networking solution for discovery services

APIs is used for canary deployment. There are two different helm chart configurations implemented for this application one for supporting Kubernetes gateway API and Istio service mesh. The other for supporting Istio service mesh without Kubernetes gateway API. Each helm chart defined consists of two deployments - productionDeployment (replica count 2) and canaryDeployment (replica count 2). Kubernetes service definition, Istio destination rule and Istio virtual service are defined for both stable and canary service versions of the application. The microservice 'HelloStats' can auto scale by increasing the number of maximum and minimum replicas based on compute utilization and requests per seconds threshold. It is always advisable to run multiple instances of pods to ensure that there is enough capacity to serve requests. The environment configuration applied for this exercise include Port Forward to add istio-injection enabled labels to the 'HelloStats' application namespaces. These configurations are applied for the Istio proxy container sidecars to be injected in runtime to properly route traffic and balance requests weight per configuration between the canary and the stable service versions of the application.

The concurrency level is set to 10 for both the runs and 1000 HTTP requests are sent to the stable version at the start of the test and then the traffic is moved in the intervals of 10%, 40%, 90%, 100 % from stable to canary version. The Fig. 13 below shows the movement of traffic from stable to canary version. Once, the canary version is completely validated with 100% traffic, the stable version is upgraded to V2 using rolling updates and all the traffic is moved to the new upgraded stable version.

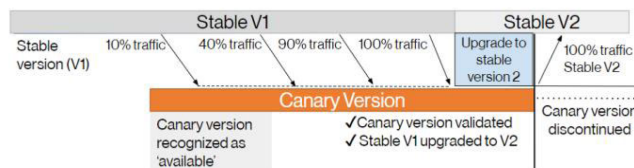

FIGURE 13. Workload moving from stable to canary version.

A. IMPLEMENTATION AND RESULTS

The reference architecture for Option 1 includes setting up the Kubernetes gateway API and service mesh as explained in Fig. 14 below.

The Table 4 below has the configurations for the canary deployment executed using Kubernetes gateway API and Istio service mesh. Table 5 captures the pipeline phases for the stable version V1 upgraded to the stable version V2 using Kubernetes gateway API and Istio service mesh.

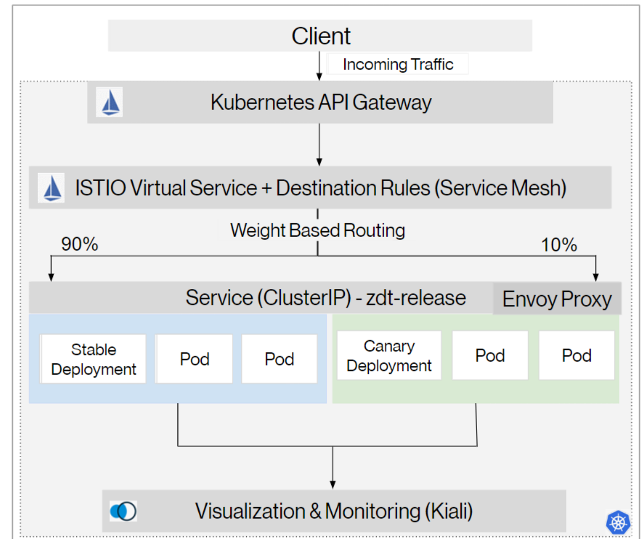

FIGURE 14. Canary deployment using kubernetes gateway API and service mesh.

TABLE 4. Configuration for kubernetes gateway API and Istio.

Parameters	Stable Version	Canary Version
gateway selector:	istio:ingress match: --headers: --weights: Hosts: "*"	istio:ingress match: --headers: --weights: Hosts: "*"
Virtual service gateways:	name: production labels: Canary:false tag name: v1 labels:version:v1(stable)	name:canary labels: Canary:true tag name: v2 labels: version:v2(canary)

The metrics of the test run executed using Kubernetes gateway API and Istio as service mesh are captured in the Table 6 below.

The second run for the canary deployment is set up with only Istio service mesh. The workloads are moved from stable to canary version leveraging the service mesh setup. The reference architecture for option 2 with Istio service mesh is provided in the Fig. 15 below

The Table 7 below has the configurations for the canary deployment executed using Istio service mesh. The Table 8 captures the pipeline phases for the stable version V1 upgraded to the stable version V2 using Istio service mesh.

The metrics of the second test run executed using Istio service mesh are captured in the Table 9 below:

B. OBSERVATIONS

As next steps to conclude the findings, the timings collected from both the runs mentioned in Table 6 and Table 9 is plotted

TABLE 5. Pipeline phases for kubernetes gateway API and Istio.

Phase	Stable version	Canary version
STEADY STATE 100	100% → V1 (2) Replicas	0% → V1 (0) Replicas
CANARY 10	90% → V1 (2) Replicas	10% → V2 (2) Replicas
CANARY 40	60% → V1 (2) Replicas	40% → V2 (2) Replicas
CANARY 90	10% → V1 (2) Replicas	90% → V2 (2) Replicas
CANARY 100	0% → V1 (2) Replicas	100% → V2 (2) Replicas
CANARY 100 ROLLING UPGRADE	0% → V2	100% → V2
PROD 100	100% → V2 (2) Replicas	0% → V2 (2) Replicas
NEW STEADY STATE PROD 100	100% → V2 (2) Replicas	0% → V2 (0) Replicas
ROLLBACK	100% → V1 (2) Replicas	0% → V1 (0) Replicas

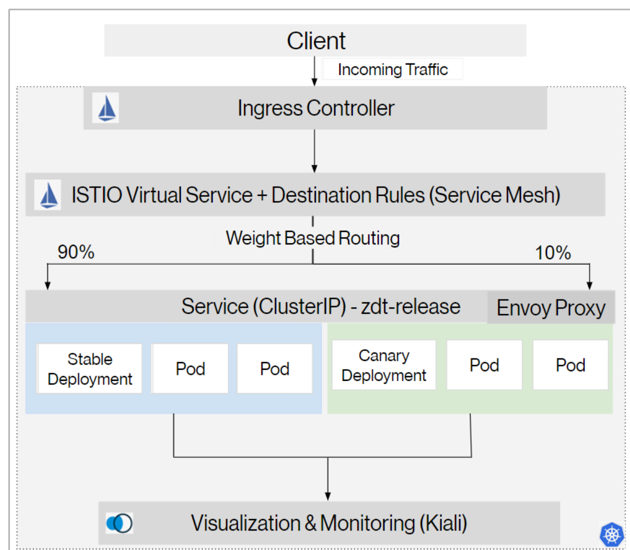

FIGURE 15. Canary deployment using service mesh.

TABLE 6. Metrics for kubernetes gateway API and Istio.

Parameters	Metrics
State	**Stable – 100%; Canary – 0%**
Time taken for tests	152.024 seconds
Request per second	6.58 [#/sec] (mean)
Time per request	1520.241 [ms] (mean)
Time per concurrent requests	152.024 [ms]
Failed requests	0 out of 1000
State	**Stable - 90%; Canary - 10%**
Time taken for tests	151.944 seconds
Request per second	6.58 [#/sec] (mean)
Time per request	1519.437 [ms] (mean)
Time per concurrent requests	151.944 [ms]
Failed requests	0 out of 1000
State	**Stable - 60%; Canary - 40%**
Time taken for tests	151.857 seconds
Request per second	6.59 [#/sec] (mean)
Time per request	1518.56 [ms] (mean)
Time per concurrent requests	151.857 [ms]
Failed requests	0 out of 1000
State	**Stable - 10%; Canary - 90%**
Time taken for tests	152.009 seconds
Request per second	6.59 [#/sec] (mean)
Time per request	1520.085 [ms] (mean)
Time per concurrent requests	152.009 [ms]
Failed requests	0 out of 1000
State	**Stable - 0%; Canary - 100%**
Time taken for tests	151.988 seconds
Request per second	6.58 [#/sec] (mean)
Time per request	1519.884 [ms] (mean)
Time per concurrent requests	151.988 [ms]
Failed requests	0 out of 1000
State	**Stable upgraded to v2, Canary 100%**
Time taken for tests	152.037 seconds
Request per second	6.57 [#/sec] (mean)
Time per request	1520.366 [ms] (mean)
Time per concurrent requests	152.037[ms]
Failed requests	0 out of 1000
State	**Stable 100%, Canary - backup**
Time taken for tests	152.271 seconds
Request per second	6.57 [#/sec] (mean)
Time per request	1522.706 [ms] (mean)
Time per concurrent requests	152.271 [ms]
Failed requests	0 out of 1000
State	**Stable 100%, Canary -0 replicas**
Time taken for tests	152.271 seconds
Request per second	6.57 [#/sec] (mean)
Time per request	1521.685 [ms] (mean)
Time per concurrent requests	152.168 [ms]
Failed requests	0 out of 1000

in the graph shown in Fig. 16 below. The graph shows the metrics collected for the time taken for tests, time per request and time per concurrent requests. These numbers do not show any vast variation between the two techniques. The other important criteria considered is number of failed requests. For both the test runs completed using different techniques, there were no failed or interrupted requests. It was observed that the number of requests handled per second increased or doubled when the implementation had only service mesh. For the test run using Kubernetes gateway API and service mesh, the implementation handled approx. 6.58 requests per second and then for the test run using service mesh alone, the implementation handled approx. 13.15 requests per second.

The two test runs were completed using weightage-based routing. To accomplish this, 'DestinationRule'is defined to provide the subset version of the application - v1 and v2. Next, to distribute the traffic 'VirtualService' has the weights specified to move the workloads between v1 and v2.

An application can opt for either of two options explained here for the test run. It is evident that there is not a major variation from a metrics perspective so choosing the option that works best will be driven by the nature of the application architecture. Kubernetes gateway API is responsible for managing the ingress and egress traffic from external

TABLE 7. Configuration for Istio service mesh setup.

Parameters	Stable Version	Canary Version
gateway selector:	gateway selector – Not Applicable match:headers & weights	gateway selector – Not Applicable match:headers & weights
Virtual service gateways:	Hosts: Not Applicable name: production labels: Canary:false tag name: v1 labels:version:v1(stable)	Hosts: Not Applicable name:canary labels:Canary:true tag name: v2 labels: version:v2(canary)

TABLE 8. Pipeline phases For Istio service mesh.

Phase	Stable version	Canary version
STEADY STATE 100	100% → V1 (2) Replicas	0% → V1 (0) Replicas
CANARY 10	90% → V1 (2) Replicas	10% → V2 (2) Replicas
CANARY 40	60% → V1 (2) Replicas	40% → V2 (2) Replicas
CANARY 90	10% → V1 (2) Replicas	90% → V2 (2) Replicas
CANARY 100	0% → V1 (2) Replicas	100% → V2 (2) Replicas
CANARY 100 ROLLING UPGRADE	0% → V2	100% → V2
PROD 100	100% → V2 (2) Replicas	0% → V2 (2) Replicas
NEW STEADY STATE PROD 100	100% → V2 (2) Replicas	0% → V2 (0) Replicas
ROLLBACK	100% → V1 (2) Replicas	0% → V1 (0) Replicas

sources. On the other hand, service mesh is responsible for internal traffic between the services (east-west). When only Istio service mesh is used, there is an in-built ingress API gateway for external traffic that can be leveraged. If the application is looking for simpler architecture with lesser hand-offs, then leveraging only Istio service mesh is doable, however the application will not be able to take advantage of the additional features like logging, observability and tracking that are offered by Kubernetes gateway API. In the test run, the number of requests handled per second were double when we only had Istio service mesh and it reduced by 50% when we had Kubernetes gateway API introduced. However, if the application is keen on decoupling the functions for managing ingress and egress traffic then Kubernetes gateway API should be set up so there is a clear separation of functionality. This will increase the maintenance for the application; however, the application will be able to better leverage the additional features.

The test runs conducted in this section were limited to service changes and the complexity increases with the introduction of the database schema changes. The technique presented in the next section elaborates on how the test runs conducted can be executed to include both services and database schema changes. The next section also provides a complete reference architecture for the canary deployments.

TABLE 9. Metrics for Istio service mesh.

Parameters	Metrics
State	**Stable – 100%; Canary – 0%**
Time taken for tests	76.049 seconds
Request per second	13.15 [#/sec] (mean)
Time per request	760.494 [ms] (mean)
Time per concurrent requests	76.049 [ms]
Failed requests	0 out of 1000
State	**Stable - 90%; Canary - 10%**
Time taken for tests	75.583 seconds
Request per second	13.23 [#/sec] (mean)
Time per request	755.827 [ms] (mean)
Time per concurrent requests	75.583 [ms]
Failed requests	0 out of 1000
State	**Stable - 60%; Canary - 40%**
Time taken for tests	70.387 seconds
Request per second	14.21 [#/sec] (mean)
Time per request	703.867 [ms] (mean)
Time per concurrent requests	70.387 [ms]
Failed requests	0 out of 1000
State	**Stable - 10%; Canary - 90%**
Time taken for tests	67.028 seconds
Request per second	14.92 [#/sec] (mean)
Time per request	670.280 [ms] (mean)
Time per concurrent requests	67.028 [ms]
Failed requests	0 out of 1000
State	**Stable - 0%; Canary - 100%**
Time taken for tests	65.512 seconds
Request per second	15.26 [#/sec] (mean)
Time per request	655.121 [ms] (mean)
Time per concurrent requests	65.512 [ms]
Failed requests	0 out of 1000
State	**Stable upgraded to v2, Canary 100%**
Time taken for tests	76.527 seconds
Request per second	13.07 [#/sec] (mean)
Time per request	765.268 [ms] (mean)
Time per concurrent requests	76.527 [ms]
Failed requests	0 out of 1000
State	**Stable 100%, Canary - backup**
Time taken for tests	80.575 seconds
Request per second	12.41 [#/sec] (mean)
Time per request	805.575 [ms] (mean)
Time per concurrent requests	80.575 [ms]
Failed requests	0 out of 1000
State	**Stable 100%, Canary -0 replicas**
Time taken for tests	73.712 seconds
Request per second	13.57 [#/sec] (mean)
Time per request	737.125 [ms] (mean)
Time per concurrent requests	73.712 [ms]
Failed requests	0 out of 1000

VII. CANARY WITH DATABASE CHANGES

Applications built using the microservices architecture leverage relational databases. Any change to the table structure, table definitions and schema requires execution of the statements using DDL (Data Driven Language) and this leads to downtime. So, while multiple deployment strategies have addressed the challenges of maintaining zero downtime for service changes, there is limited work done to

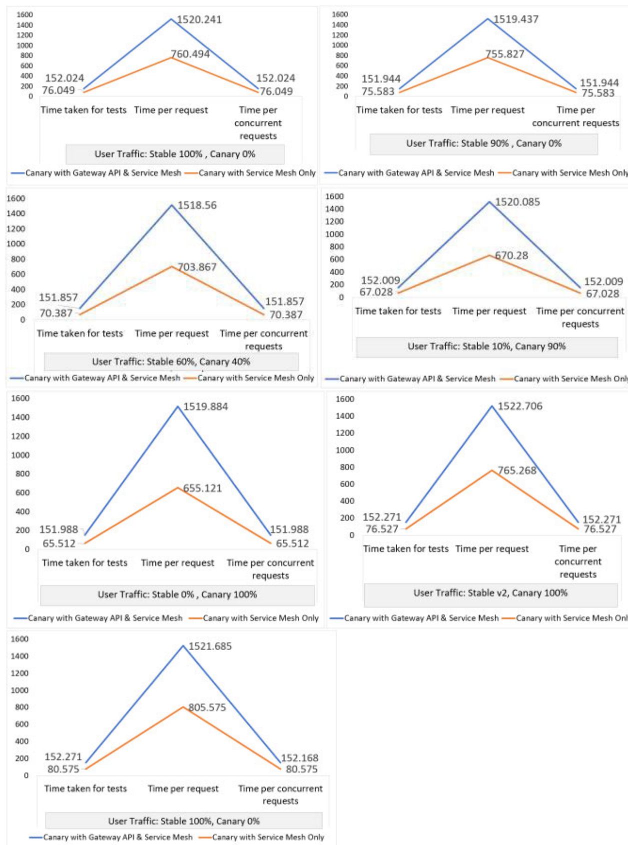

FIGURE 16. Graph with metrics for Canary deployments with Kubernetes Gateway API and Service Mesh and with Service Mesh alone.

achieve zero downtime for database schema changes. In this section of the paper, we will elaborate further on the canary deployment technique and use Liquibase, Istio and load balancer to perform database schema changes without any downtime.

A. ENVIRONMENT SETUP AND IMPLEMENTATION

The technology stack that is used in the implementation includes Kubernetes for hosting multiple versions of applications as containers. Ingress to balance the traffic weightages across the multiple versions of the application to realize canary deployment pattern and Istio service mesh. Liquibase to manage different versions of the database schema sets. Liquibase is an open-source tool used for tracking, logging and managing database changes as explained in [43] and [44].

Liquibase is leveraged to support Database Change Management. The property value 'expand' for the Liquibase controller is used to introduce new schema changes without removing the old changes. The old schema changes will be removed when services are deployed with the 'contract' property value and when 100% of the traffic goes to the new version of the service. The intricacies of performing database schema changes are further explained in [45] and [46].

The technique aims to achieve zero downtime when both services and database schema changes have to be deployed as a single transaction. The microservice code fully manages the database schema. Application code and schema changesets are deployed in a single repository for better service maintenance and this also eliminates the need for separate database support or specific deployment pipelines to manage database change deployment in a silo.

The technique leverages the concept of 'contexts' supported by Liquibase to distinguish the expanded state of the application where both old and new schema changes have to exist together to serve both stable and canary versions of the application during the canary deployment phases. The technique explained here uses Liquibase to manage the different versions of the database. The 'changesets' are used in this implementation to manage database version control. Changesets are defined in a changelog as a collection of individual changes which are used to track and apply database schema changes. The microservice code uses the Liquibase maven plugin to perform rollout and rollback operation to a specific version tag. The 'context' property value is sent as a JVM argument at the start of the application and the Liquibase changeset files are marked with context as 'contract' or 'expand' attribute. The changesets will be executed depending on the context value that matches the different stages of the deployment pipeline Also, the implementation utilizes a spring configuration class called Liquibase configuration with a task executor to ensure that the microservice startup fails if Liquibase changes runs into integrity issues and are not able to successfully perform the required changes. This enables early recognition of any potential deployment issues related to the database.

To execute the technique, the microservice version V1 is deployed as step 1. Liquibase executes the database changesets associated with the version V1 of the code as the microservice comes up on the Kubernetes cluster. This represents the stable deployment of the microservice version V1. In step 2, microservice version V2 is deployed as a canary release. Liquibase will maintain the old and new schema associated with the microservice version V1 and V2. This will enable the microservice version V1 to remain up and running servicing a portion of the segmented traffic. The microservice version V2 is deployed with Liquibase context property to expand so that version V1 of the application does not break due to the introduction of the new schema changes. In step 3, the new microservice version V2 introduced as a canary release is validated and once certified, step 4 is executed where the user traffic is gradually shifted to the new microservice version V2. In step 5, the microservice version V1 is phased out and microservice version V2 is deployed as the new stable version. The Liquibase context property is updated as 'contract' and this will trigger the underlying database changesets that will do the necessary cleanup to remove the changesets in place to serve microservice version V1. Finally, in step 6 the complete user traffic

is transitioned to the new stable application version V2. This entire technique employs an expand and contract strategy for managing database schema changes with Liquibase handling the different database versions.

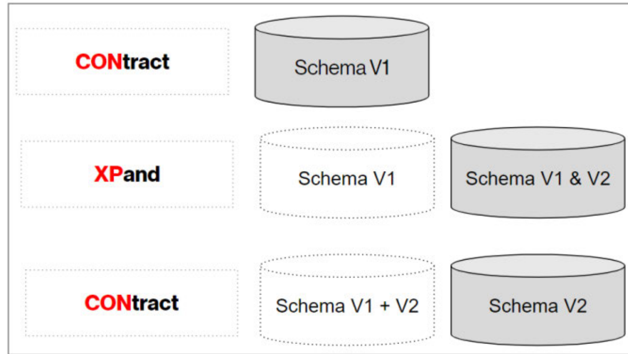

FIGURE 17. Illustration of the Contract/Expand method.

The Fig. 17 below provides a visual representation of the contract and expand pattern and Table 10 maps the deployment phases to the contract and expand pattern.

TABLE 10. Deployment pipeline phases correlated with liquibase.

Phase	Stable version	Canary version	Schema Version / State
STEADY STATE 100	100% → V1 (2) Replicas	0% → V1 (0) Replicas	V1 (Contract)
CANARY 10	90% → V1 (2) Replicas	10% → V2 (2) Replicas	V1 +V2 (Expand)
CANARY 40	60% → V1 (2) Replicas	40% → V2 (2) Replicas	V1 +V2 (Expand)
CANARY 90	10% → V1 (2) Replicas	90% → V2 (2) Replicas	V1 +V2 (Expand)
CANARY 100	0% → V1 (2) Replicas	100% → V2 (2) Replicas	V1 +V2 (Expand)
CANARY 100 ROLLING UPGRADE	0% → V2	100% → V2	V1 +V2 (Expand)
PROD 100	100% → V2 (2) Replicas	0% → V2 (2) Replicas	V2 (Contract)
NEW STEADY STATE PROD 100	100% → V2 (2) Replicas	0% → V2 (0) Replicas	V2 (Contract)
ROLLBACK	100% → V1 (2) Replicas	0% → V1 (0) Replicas	V1 (Contract)

In the previous section, the paper evaluated techniques to perform canary deployments for services changes using Kubernetes gateway API and Istio service mesh. In this section, the paper elaborates the technique to manage database schema changes along with services changes without incurring any application downtime. The paper shows how uniquely the expand and contract pattern can be extended for canary deployments leveraging tools like load balancer and Liquibase for database version management. The complete reference architecture for canary deployment is provided in Fig. 18 below.

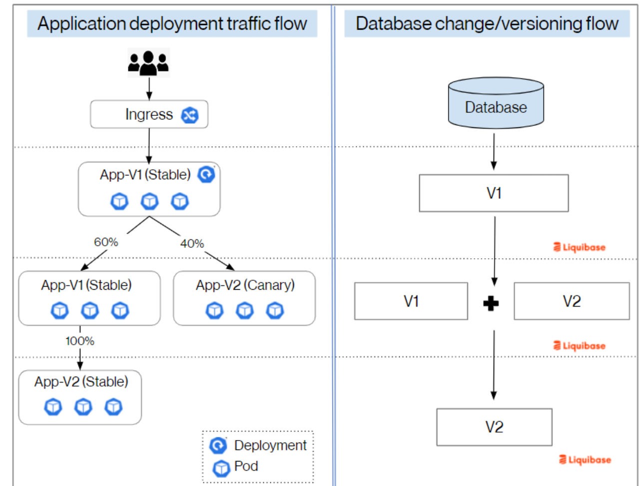

FIGURE 18. Reference architecture for Canary deployments including services and database schema changes.

B. ZERO DOWNTIME COMPLMENTING PRIVACY AND SECURITY FOR CLOUD COMPUTING

The zero downtime technique discussed in this paper can be leveraged by enterprise organizations for software development and delivery. In addition, the zero downtime technique can also complement and enhance the effectiveness of recent schemes like privacy-preserving reputation updating (PPRU) scheme for cloud-assisted vehicular networks [47], time-controllable keyword search scheme [48] and privacy-preserving spatial data query in cloud computing [49]. Zero downtime deployment can enable continuous updates to the privacy-preserving reputation updating (PPRU) system without interrupting its operation. The reputation scores can be updated without incurring any downtime and this enhances the reliability and accuracy of the reputation system in vehicular networks. Leveraging the canary deployment technique discussed in this paper, the privacy-preserving reputation updating (PPRU) system can be updated gradually and any potential issues or vulnerabilities can be identified and addressed early, minimizing the risk of failures in the process. Similarly, zero downtime practices can facilitate iterative improvements to the time-controllable keyword search scheme which is designed for secure and privacy-preserving keyword search operations. This functionality helps in accessing medical records and documents or information stored in the cloud repository. Real-time dynamic policy updates can be made to enhance the responsiveness of the scheme to changing user preferences and access rights. Similarly, continuous optimizations can be applied to privacy preserving spatial data query in cloud computing. These queries contain sensitive location information and allow organizations to analyze spatial data without revealing precise location details of individuals. The zero downtime technique discussed in this paper can help to keep the query latest and be able to apply dynamic updates with incurring any data or service loss.

The schemes discussed in this section are used for maintaining the privacy and security for critical domains like vehicular network and health care. These schemes need regular updates and any downtime may lead to loss of data that otherwise needs to be dynamically updated in real-time Hence, the zero downtime technique discussed in this paper can complement these schemes in a cloud computing environment.

VIII. CONCLUSION

The paper summarizes the building blocks necessary for applications to support zero downtime code deployments. The paper provides a qualitative assessment of the different code deployment techniques that the business can use to identify the right strategy as per the use case and business requirements. Based on the qualitative assessment, the paper dives deeper on the canary deployment techniques. The paper performs evaluation of the canary deployment technique for the service changes using Kubernetes gateway API and Istio service mesh. It is evident from the evaluation that either of the techniques can be applied and decision for choosing between the two techniques should be driven by the application architecture and whether the business functions require to use the additional capabilities offered by the Kubernetes gateway API like logging and traceability. Kubernetes in-built features are not adequate to achieve true zero downtime for deployments that involve both services and database schema changes. Hence, the paper takes canary deployment a step further and covers how zero downtime deployment for service and database schema changes can be achieved. The novel technique implemented in this paper shows how the concept of 'contract' and 'expand' can be used along with tools like Liquibase, Istio and load balancer to achieve zero downtime code deployments for services and database schema changes. With this technique, the paper mitigates the technical challenge in achieving zero downtime for services and database schema changes. The technique discussed in this paper enables zero downtime code deployment for both services and database related changes.

The paper covers the various aspects of the canary deployment to provide a complete reference architecture. The paper elaborates on the canary deployment decision workflow and shows that canary deployments offer the flexibility to easily rollback the changes if any issues are encountered with the new version. Canary deployments can be implemented without running multiple parallel environments and this can also help in lowering the operational expense for organizations working towards zero downtime architecture for code deployments. The paper has not covered the integration of the technique explained here with the deployment pipeline. It will be ideal to have a single step dedicated in the pipeline process for promoting the database schema changes, however this paper does not capture that process in detail. Enterprise organizations can use this paper as a reference architecture to build the process related to canary deployments for achieving zero downtime code deployments.

ACKNOWLEDGMENT

The authors would like to acknowledge the contribution of these team members toward the study and implementation of best practices for high availability and near zero downtime experience. The study related to zero downtime code deployments for cloud native enterprise applications was conducted as part of the Verizon initiative to build resilient, highly available, and cloud native applications under the leadership of Sebastien Jobert [Executive Director with Verizon (e-mail: sebastien.jobert@verizon.com)] and Ashuma Singh Kaul [Executive Director with Verizon (e-mail: ashuma.s.kaul@verizon.com)]. Application Development Engineer: Deepak Kumar; DevOps Engineers: Yaswanth Nadella and Ruban Sathyamoorthy; and Verizon India Partners: Saranya Kumaraguruparan and Ritu Sharma.

REFERENCES

[1] M. Tuovinen. (2023). *Reducing Downtime During Software Deployment*. Open Access Res. Papers. Accessed: Nov. 27, 2023. [Online]. Available: https://core.ac.uk/download/pdf/250163188.pdf

[2] K. Gallaba, "Improving the robustness and efficiency of continuous integration and deployment," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)*, Sep. 2019, pp. 619–623, doi: 10.1109/ICSME.2019.00099.

[3] M. Meyer, "Continuous integration and its tools," *IEEE Softw.*, vol. 31, no. 3, pp. 14–16, May 2014, doi: 10.1109/MS.2014.58.

[4] S. Bobrovskis and A. Jurenoks. (2023). *A Survey of Continuous Integration, Continuous Delivery and Continuous.* Accessed: Nov. 26, 2023. [Online]. Available: https://ceur-ws.org/Vol-2218/paper31.pdf

[5] B. Adams, S. Bellomo, C. Bird, T. Marshall-Keim, F. Khomh, and K. Moir, "The practice and future of release engineering: A roundtable with three release engineers," *IEEE Softw.*, vol. 32, no. 2, pp. 42–49, Mar. 2015, doi: 10.1109/MS.2015.52.

[6] T. Savor, M. Douglas, M. Gentili, L. Williams, K. Beck, and M. Stumm, "Continuous deployment at Facebook and OANDA," in *Proc. 38th Int. Conf. Softw. Eng. Companion*, May 2016, doi: 10.1145/2889160.2889223.

[7] C. K. Rudrabhatla, "Comparison of zero downtime based deployment techniques in public cloud infrastructure," in *Proc. 4th Int. Conf. I-SMAC, IoT Social, Mobile, Anal. Cloud (I-SMAC)*, Oct. 2020, pp. 1082–1086, doi: 10.1109/I-SMAC49090.2020.9243605.

[8] B. Yang, A. Sailer, S. Jain, A. E. Tomala-Reyes, M. Singh, and A. Ramnath, "Service discovery based blue-green deployment technique in cloud native environments," in *Proc. IEEE Int. Conf. Services Comput. (SCC)*, Jul. 2018, pp. 185–192, doi: 10.1109/SCC.2018.00031.

[9] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. Reading, MA, USA: Addison-Wesley, 2015.

[10] A. Tarvo, P. F. Sweeney, N. Mitchell, V. T. Rajan, M. Arnold, and I. Baldini, "CanaryAdvisor: A statistical-based tool for Canary testing (demo)," in *Proc. Int. Symp. Softw. Test. Anal.*, Jul. 2015, pp. 418–422, doi: 10.1145/2771783.2784770.

[11] D. Ernst, A. Becker, and S. Tai, "Rapid Canary assessment through proxying and two-stage load balancing," in *Proc. IEEE Int. Conf. Softw. Archit. Companion (ICSA-C)*, Mar. 2019, pp. 116–122, doi: 10.1109/ICSA-C.2019.00028.

[12] N. Strasser. (2023). *An Evaluation of Canary Deployment Tools*. Accessed: Nov. 27, 2023. [Online]. Available: https://pub.fh-campuswien.ac.at/obvfcwhsacc/content/titleinfo/8874850/full.pdf

[13] V. Sharma, "Managing multi-cloud deployments on kubernetes with istio, prometheus and grafana," in *Proc. 8th Int. Conf. Adv. Comput. Commun. Syst. (ICACCS)*, vol. 1, Mar. 2022, pp. 525–529, doi: 10.1109/ICACCS54159.2022.9785124.

[14] G. Booch, "Object-oriented development," *IEEE Trans. Softw. Eng.*, vol. SE-12, no. 2, pp. 211–221, Feb. 1986, doi: 10.1109/TSE.1986.6312937.

[15] K. Beck and C. Andres, *Extreme Programming Explained: Embrace Change*. Reading, MA, USA: Addison-Wesley, 2012.

[16] H. Lei, F. Ganjeizadeh, P. K. Jayachandran, and P. Ozcan, "A statistical analysis of the effects of scrum and Kanban on software development projects," *Robot. Comput.-Integr. Manuf.*, vol. 43, pp. 59–67, Feb. 2017, doi: 10.1016/j.rcim.2015.12.001.

[17] J. Hall. (2021). *A Brief History of CI/CD*. Jonathan Hall Blog. Accessed: Nov. 27, 2023. [Online]. Available: https://jhall.io/archive/2021/09/26/a-brief-history-of-ci/cd/

[18] G. O'Regan, "History of software engineering," in *A Brief History of Computing*. NY, USA: Springer, 2021, pp. 201–225, doi: 10.1007/978-3-030-66599-9_16c.

[19] A. Malhotra, A. Elsayed, R. Torres, and S. Venkatraman, "Evaluate solutions for achieving high availability or near zero downtime for cloud native enterprise applications," *IEEE Access*, vol. 11, pp. 85384–85394, 2023, doi: 10.1109/ACCESS.2023.3303430.

[20] K. J. Sullivan, W. G. Griswold, Y. Cai, and B. Hallen, "The structure and value of modularity in software design," in *Proc. 8th Eur. Softw. Eng. Conf. 9th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, Sep. 2001, pp. 99–108, doi: 10.1145/503209.503224.

[21] L. De Lauretis, "From monolithic architecture to microservices architecture," in *Proc. IEEE Int. Symp. Softw. Rel. Eng. Workshops (ISSREW)*, Oct. 2019, pp. 93–96, doi: 10.1109/ISSREW.2019.00050.

[22] S. Hardikar, P. Ahirwar, and S. Rajan, "Containerization: Cloud computing based inspiration technology for adoption through Docker and kubernetes," in *Proc. 2nd Int. Conf. Electron. Sustain. Commun. Syst. (ICESC)*, Aug. 2021, pp. 1996–2003, doi: 10.1109/ICESC51422.2021.9532917.

[23] V. G. D. Silva, M. Kirikova, and G. Alksnis, "Containers for virtualization: An overview," *Appl. Comput. Syst.*, vol. 23, no. 1, pp. 21–27, May 2018.

[24] M. Aleksic. (2023). *Containerization vs. Virtualization? Understand the Differences*. Accessed: Nov. 27, 2023. [Online]. Available: https://ubuntu.com/blog/containerization-vs-virtualization

[25] J. Watada, A. Roy, R. Kadikar, H. Pham, and B. Xu, "Emerging trends, techniques and open issues of containerization: A review," *IEEE Access*, vol. 7, pp. 152443–152472, 2019, doi: 10.1109/ACCESS.2019.2945930.

[26] (2023). *What's the Difference Between Containers and Virtual Machines?* Accessed: Nov. 27, 2023. [Online]. Available: https://aws.amazon.com/compare/the-difference-between-containers-and-virtual-machines/

[27] *Azure Fundamentals: Exam AZ-900*, 30 Bird Media, Rochester, NY, USA, 2020.

[28] Y. Zhang. (2023). *Graceful Shutdown Services in Kubernetes*. Thoughtworks. Accessed: Nov. 27, 2023. [Online]. Available: https://www.thoughtworks.com/en-us/insights/blog/cloud/shutdown-services-kubernetes

[29] P. Welch. *Graceful Termination—Graceful Resetting*. Accessed: Feb. 3, 2024. [Online]. Available: https://kar.kent.ac.uk/20953/1/GracefulWelch.pdf

[30] Y. Zhang. *Graceful Shutdown Services in Kubernetes*. Accessed: Nov. 27, 2023. [Online]. Available: https://www.thoughtworks.com/en-us/insights/blog/cloud/shutdown-services-kubernetes

[31] Harness Developer Hub. (2023). *Graceful Delegate Shutdown*. Accessed: Nov. 27, 2023. [Online]. Available: https://developer.harness.io/docs/platform/delegates/delegate-concepts/graceful-delegate-shutdown-process/

[32] M. Rahman, S. Iqbal, and J. Gao, "Load balancer as a service in cloud computing," in *Proc. IEEE 8th Int. Symp. Service Oriented Syst. Eng.*, Apr. 2014, pp. 204–211, doi: 10.1109/SOSE.2014.31.

[33] N. Malik and A. Malik, "Survey of load balancing algorithms and performance evaluation in cloud computing," in *Proc. Int. Conf. Commun. Comput. Syst. (ICCCS)*, Nov. 2016, doi: 10.1201/9781315364094-195.

[34] J. Rodela. (2023). *What is Dynamic Routing, and How Does it Work?* Accessed: Nov. 26, 2023. [Online]. Available: https://gomotive.com/blog/what-is-dynamic-routing/

[35] G. Menachem. *Kubernetes Deployment: How it Works & 5 Deployment Strategies*. Accessed: Nov. 27, 2023. [Online]. Available: https://komodor.com/learn/kubernetes-deployment-how-it-works-and-5-deployment-strategies/

[36] J. Domingus and J. Arundel, *Cloud Native Devops With Kubernetes: Building, Deploying, and Scaling Modern Applications in the Cloud*. Beijing, China: O'Reilly, 2022.

[37] *What is Open Source?* Accessed: Nov. 27, 2023. [Online]. Available: https://opensource.com/resources/what-open-source

[38] F. P. Deek and J. A. MacHugh, *Open Source: Technology and Policy*. Cambridge, U.K.: Cambridge Univ. Press, 2010.

[39] G. Sayfan, *Mastering Kubernetes Master the Art of Container Management By Using the Power of Kubernetes*. Birmingham, AL, USA: Packt Publishing, 2018.

[40] (2023). *ISTIO*. Accessed: Nov. 2, 2023. [Online]. Available: https://istio.io/

[41] Google. *What is ISTIO?* Accessed: Nov. 2, 2023. [Online]. Available: https://cloud.google.com/learn/what-is-istio

[42] Apache HTTP server benchmarking tool. (2023). *Apache HTTP Server Benchmarking Tool*. Accessed: Nov. 2, 2023. [Online]. Available: https://httpd.apache.org/docs/2.4/programs/ab.html

[43] PETE. *The LIQUIBASE Community: The Database DevOps Community*. Accessed: Nov. 27, 2023. [Online]. Available: http://www.liquibase.org/

[44] *Liquibase: Database Change Management & CI/CD Automation | Database DevOps*. Accessed: Nov. 27, 2023. [Online]. Available: http://www.liquibase.com/

[45] J.-J. Dijkstra. *Zero-Downtime Schema Changes*. Accessed: Nov. 27, 2023. [Online]. Available: https://research.utwente.nl/en/publications/zero-downtime-schema-changes

[46] C. Kampen. *Zero Downtime Schema Migrations in Highly Available Databases*. Accessed: Nov. 27, 2023. [Online]. Available: https://essay.utwente.nl/92098/1/vanKampen_MA_EEMCS.pdf

[47] Z. Liu et al., "PPRU: A privacy-preserving reputation updating scheme for cloud-assisted vehicular networks," *IEEE Trans. Veh. Technol.*, pp. 1–16, 2024, doi: 10.1109/tvt.2023.3340723.

[48] Y. Miao, F. Li, X. Li, Z. Liu, J. Ning, H. Li, K. R. Choo, and R. H. Deng, "Time-controllable keyword search scheme with efficient revocation in mobile E-health cloud," *IEEE Trans. Mobile Comput.*, vol. 23, no. 5, pp. 3650–3665, May 2024, doi: 10.1109/TMC.2023.3277702.

[49] Y. Miao, Y. Yang, X. Li, L. Wei, Z. Liu, and R. H. Deng, "Efficient privacy-preserving spatial data query in cloud computing," *IEEE Trans. Knowl. Data Eng.*, vol. 36, no. 1, pp. 122–136, Jan. 2024, doi: 10.1109/TKDE.2023.3283020.

[50] R. Jeyaraj, A. Balasubramaniam, N. Guizani, and A. Paul, "Resource management in cloud and cloud-influenced technologies for Internet of Things applications," *ACM Comput. Surv.*, vol. 55, no. 12, pp. 1–37, Mar. 2023, doi: 10.1145/3571729.

ANTRA MALHOTRA (Senior Member, IEEE) was born in Delhi, India. She received the first master's degree in business and finance from Mumbai University, Mumbai, India, in 2006, and the second master's degree in computer science from Georgia Institute of Technology, Atlanta, USA, in 2022.

She is currently with Verizon Network Systems Team, Temple Terrace, FL, USA, as an Associate Director–Software Development. She is responsible for the application development and software delivery for the big data platforms that provide intelligent location services, product availability, and network data collection framework. Prior to joining Verizon, she was with Hutchison 3G U.K., Mumbai, India; and Acclaris (product based company), Tampa, FL. She is a Guest Faculty with the Hillsborough Community College–ICCE for teaching cloud concepts and technologies. Her area of research and work involves cloud technologies, application stability, and big data management.

AMR ELSAYED was born in Cairo, Egypt.

He is a technology geek. He currently holds the position of a Distinguished Engineer–Data Science with Verizon, Temple Terrace, FL, USA. He plays a pivotal role in providing innovative solutions across Verizon's Network Systems. Before joining Verizon, he was with the IBM Clients Innovation Center, where he honed his skills and expertise in the technology field. His diverse talents and dedications make him a valuable asset in the technology industry and a source of inspiration for aspiring writers.

RANDOLPH TORRES was born in Miami, FL, USA. He received the Bachelor of Arts degree in science and computer science from Florida International University, USA. He is currently with Verizon Network Systems, Temple Terrace, FL, and leads the architecture and technology strategy for the Shared Platforms Organization. His responsibilities range from AI/ML modeling to complex architectures relating to high performance computing, data integration, and fault resilience. He has contributed to Open Config and holds many patents for network systems in production with Verizon.

SRINIVAS VENKATRAMAN was born in Chennai, India. He received the Master of Engineering degree from the Indian Institute of Science in Metallurgy. He is currently a AVP–Software Development, leading the Shared Platform Portfolio for Verizon, New Jersey, USA. In his current role, he manages suite of enterprise applications and tools that facilitate service assurance. He is passionate about technology and leads the technology recommendation group across network systems.