Comparison of zero-downtime-based deployment techniques

Rushikesh Mahindra Ahirrao
SIES College Of Management Studies
Department: Computer Applications
rushikesha.mca23@siescoms.sies.edu.in

Sudhir Liladhar Singh
SIES College Of Management Studies
Department: Computer Applications
sudhirs.mca23@siescoms.sies.edu.in

Abstract**—**Zero-downtime deployment has become an essential goal in modern cloud-native software systems, particularly in high-traffic environments where even minor outages can lead to user dissatisfaction and revenue loss. This paper examines key deployment techniques such as Blue-Green, Canary, and Rolling deployments, evaluating their respective trade-offs and effectiveness. Beyond traditional strategies, we explore micro-level optimizations including connection draining, automated rollback, health probes, feature flagging, and schema-aware deployment pipelines. Through a comparative simulation in Kubernetes environments with and without such optimizations, we demonstrate notable reductions in downtime and improved deployment reliability. Our findings offer a holistic framework for engineering teams seeking continuous service availability without compromising agility or resilience.

Keywords— CI/CD(Continuous Integration / Continuous Deployment), Kubernetes, Deployment DAG (Directed Acyclic Graph), High Availability, Microservices Architecture, BG(Blue-Green),

I. INTRODUCTION (*HEADING 1*)

Before the advent of public cloud platforms, majority of the web applications followed a design pattern where in the backend and front end was compiled and packed into a single monolithic unit called web archive (war) or an enterprise archive (ear) file. These files were usually deployed on the enterprise middle ware servers like WebLogic, WebSphere or JBoss to name a few, which run on the virtual machines in the local data centers. These virtual machines run on top of a bare metal server which has a host operating system and multiple guest partitions done using a software like hypervisor.

A typical monolithic deployment architecture can be represented as in Fig 2. Since the setup with virtual machines and hypervisor software is heavy, the time required to bootup the server is high. Also, with the heaviness of binaries and libraries involved in the enterprise application and web servers involved in the design, the deployment time was significantly high. In this traditional approach, a deployment would mean replacing the older version with the newer version of the application on the same infrastructure. This not only causes outages during the deployments, but also makes the rollback difficult to implement. To counter these shortfalls and solve the heaviness in the design, micro service architecture (MSA) was brough into place. This involved breaking the heavy monolith into multiple smaller, light weight and fully decoupled units. This design brought many advantages like service isolation, separation of concerns, leverage to use heterogeneous technologies, and faster defect identification and build cycles to name a few. The concept of containerization was long existent in the Unix world. Many technologies like Docker, Rocket, Odin etc. have gained a lot of traction with the advent of public cloud platforms like AWS, Azure and GCP to name a few.

The primary reason for that being, containers are super lightweight compared to virtual machines. The underpinning reason for the lightness comes from the fact that a virtual machine is an atomic unit which contains all the heavy libraries and binaries in itself whereas a container shares the common files with others, thus making it a lighter unit. These container technologies are found to be highly suitable for the micro services which are fundamentally isolated from each other. Public cloud platforms added a great support for the container deployments due to their ability to spin up the compute resources in a very short span of time. To handle the deployment and configurations of the containers in the cloud, newer orchestration technologies like Docker swarm and Kubernetes came in to light. This combination of micro service architecture, containerization, orchestration and public cloud compute resources have significantly reduced the deployment timelines. However, the modern businesses are extremely dynamic and fluid in nature. They demand changes in the web applications continuously and rapidly. Though the containerized microservices running on the public cloud infrastructure are light weight and quick to deploy, they still cause an outage when a new patch is deployed.

A zero-downtime deployment would ideally not cause any outage to the end users. The old version continues to run till the new version is ready. That way the traffic is routed seamlessly to the newer version without any disruption. To handle the deployment of newer changes without causing an outage, Blue Green (BG) deployment strategies need to be applied. There are multiple ways to achieve the BG deployment in a cloud platform. Identifying the right strategy based on the need is critical for attaining the optimal performance during the deployment. Rest of this paper is designed as follows – In section II we explain the Blue Green deployment methodology in the cloud environment.

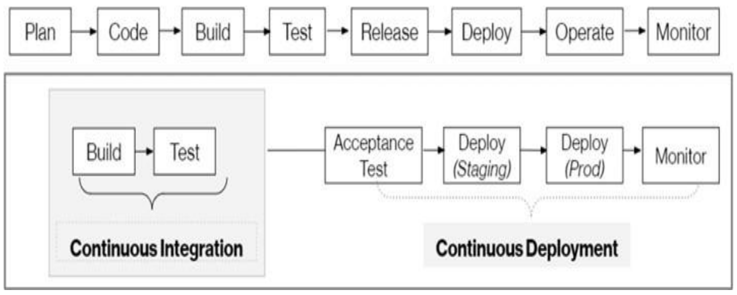


Fig- 1. Process flow of a continuous integration and continuous deployment pipeline.

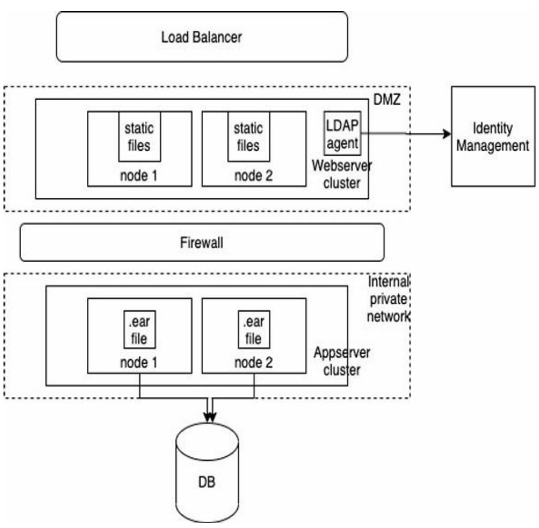


Fig. 2. Deployment architecture for a monolithic application in the local datacenter.

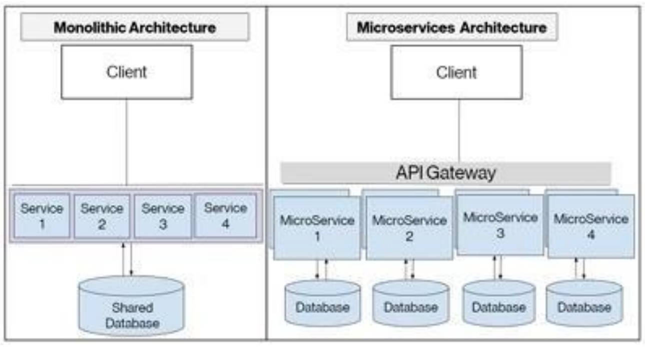


Fig. 3. A comparative view between monolithic and microservices based architecture.

II. History and Related Work

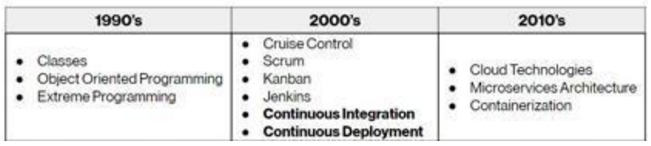
The early days of software development were associated with small and isolated teams. Code was written without much standardized practices and the software development life cycle followed a linear sequence where the phases of requirements, design, development, testing, implementation and maintenance 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 a lack of automation.

In 1991, Grady Booch advocated frequent use of classes and objects in programming to simplify software design, and this concept of objects that enabled abstraction for complex software. Grady's intent was more focused 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. 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 impact of various methodologies on software development was significant in shaping agile frameworks for software delivery. The software industry recognized the importance of delivering software more quickly; however, there were insufficient tools and automation in place to facilitate this. In 2001, the introduction of CruiseControl was a game changer as it automated builds, testing, and commits to version control systems. This advancement enabled continuous integration and helped identify integration conflicts early in the process.

During the late 2000s, the concept of continuous deployment began to emerge with tools like Jenkins. Jenkins not only supported automated testing but also automated the deployment to staging environments. Configuration management and the ability to deploy consistently and repeatedly to environments became possible with tools like Puppet and Chef.

By the mid-2010s, new applications were being developed using a microservices architecture, allowing for independent development, testing, and deployment of services. This was followed by the rise of cloud computing and containerization technologies, such as Docker, along with container orchestration platforms like Kubernetes. These technologies introduced self-managed, self-hosted code integration and deployment tools that could operate in cloud environments and integrate with services for observability and analytics.

The timeline in Fig. 1 illustrates the evolution of continuous integration and continuous deployment techniques, leading to modern containerization. The history of continuous integration, deployment, and software engineering. The latest developments in microservices architecture and containerization aim to minimize application downtime, detect issues early 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

Fig. 4. Timeline of evolution of continuous integration and continuous deployment leading to modern day containerization

In this paper, we will qualitatively evaluate these techniques based on performance, user experience, and cost. This evaluation will help organizations determine the most suitable approach for their specific business use cases.

Despite significant advancements in managing continuous integration and deployment from a technology perspective, achieving zero downtime remains a challenge, especially when application and database schema changes are involved. Many organizations struggle to perform database upgrades while maintaining zero downtime, as these changes can be sensitive, time-consuming, and may compromise data integrity if not executed properly.

This paper introduces a novel technique for implementing canary deployment using Istio, Liquibase, and a load balancer to facilitate both service and database changes. By employing this technique, applications can achieve zero downtime during code deployments. The paper also presents a complete reference architecture for canary deployments that can be adopted to ensure zero downtime for both service and database changes or upgrades

III. ZERO DOWNTIME BASED DEPLOYMENT STRATEGIES

There are 3 major ways by which a zero-downtime deployment can be achieved, namely (1) Blue Green Deployment (2) Canary Deployment (3) Rolling deployment. There are various pros and cons for each of these techniques. Sections below explain various techniques in detail.

A. *Blue Green Deployment*

Blue green deployment methodology involves replacing the container with an older image with that of a newer image without causing any downtime. This can be achieved by various design techniques in which the micro service running in a container with older image (Blue image) is replaced with the container having the newer image (Green image). This can be easily visualized from the Fig 6 where the traffic from the DNS router is routed to Blue image initially while the Green image is being deployed.

Upon the successful deployment the traffic is routed to Green instead of Blue. This strategy is useful in case of major functionality rollouts in the cloud where all the user base can be routed from old application version to the newer version. This is well suited for cloud infrastructures with container-based deployments and less suitable for data center-based deployments.

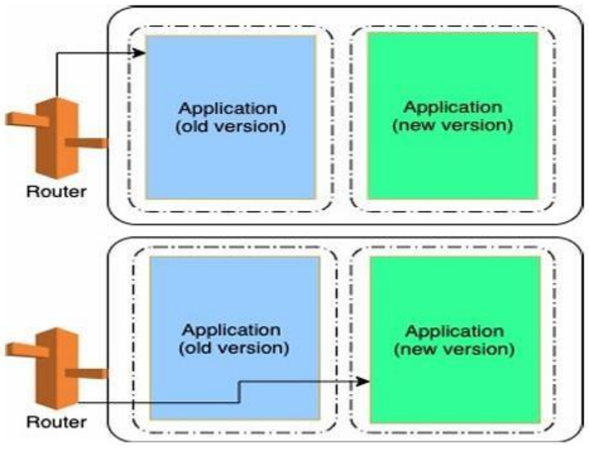

Application (old version)
Application (new version)
Router
Application (old version)
Application (new version)
Router

Fig 6. A typical Blue Green deployment diagram

B. *Canary Deployment*

Blue green deployment methodology can be performed where all the traffic is shifted from Blue version to Green version at once or it can be done even in a phased approach. This model where the traffic is routed in a phased model is called Canary style of deployment. The phased approach can be customized depending on the application needs. It can be implemented by switching the Blue version to Green to certain user population based on the type of the user or the privilege they have (or) it can be based on the geo-location of the user. For instance, the newer version may be made available to users from certain territory or region rather than rolling out the change to entire global audience. Once the green version is approved by the limited users, it is then rolled off for all the users. This strategy can be visualized from the Fig 7 below.

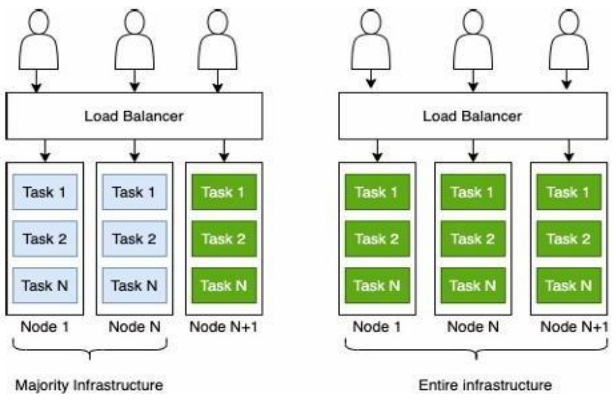


Fig 7. Blue Green deployment in Canary Style

C. *Rolling Deployment*

Rolling deployments are particularly effective for achieving zero-downtime in data center-based applications, especially when high availability is a priority. In this approach, system administrators or deployment tools sequentially upgrade subsets of application instances—ensuring that at any given point, a portion of the system remains operational with the stable version. This staged rollout minimizes risk by isolating potential failures to a smaller set of nodes, allowing quick rollback if issues are detected.

However, rolling deployments are resource-intensive and require sufficient spare capacity to handle live traffic while new versions are deployed. They work best in horizontally scalable architectures where instances can be added or removed with minimal disruption.

Despite their advantages, rolling deployments are less suitable in scenarios that demand atomic changes, such as critical security patches, breaking schema updates, or when coordinated changes across all instances are required. Additionally, managing application state, session persistence, and compatibility across mixed-version environments can complicate rollback strategies.

Automation tools like Kubernetes and Spinnaker have improved the efficiency and reliability of rolling deployments, introducing health probes and monitoring mechanisms to ensure only healthy instances are exposed to users. Nonetheless, successful execution still relies on careful planning, real-time observability, and an infrastructure setup that supports version co-existence.

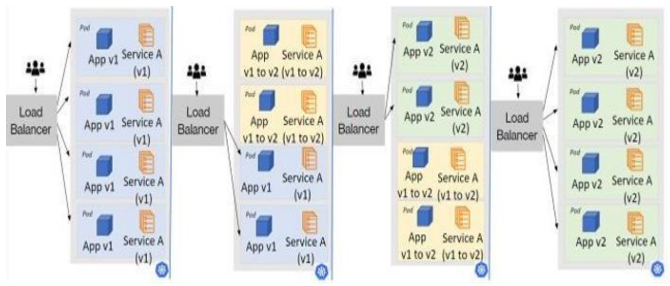


Fig. 8 Rolling deployment where the pods running with the old version of the service are replaced with the new version while continuing to service the clients.

IV. Problem definition - Scope

In today's high-availability software landscape, system downtime—even for a few seconds—can lead to significant business and reputational loss. Although Continuous Integration and Continuous Deployment (CI/CD) pipelines have enabled rapid software delivery, achieving truly zero-downtime deployment remains a complex challenge. Common problems include abrupt session termination, service inconsistency during version transitions, database migration failures, and the need for manual rollbacks. Most traditional deployment strategies either rely on over-provisioned infrastructure (e.g., Blue-Green) or introduce operational complexity (e.g., Canary or Rolling deployments) without guaranteeing zero impact on end-users. Furthermore, these approaches often neglect backend database changes, which can become single points of failure during schema upgrades.

The primary objective of this research is to analyze and improve deployment methodologies to ensure seamless software releases without service disruption. Specifically, this study aims to:

- Identify deployment bottlenecks and failure points in typical microservices environments.
- Evaluate techniques like Blue-Green, Rolling, and Canary deployments in terms of reliability, rollback efficiency, and performance impact.
- Recommend a practical framework for achieving end-to-end zero-downtime deployment in Kubernetes-based and cloud-native systems.

The scope of this research is limited to Linux-based containerized environments using Kubernetes, with simulated production traffic to reflect real-world deployment scenarios.

V. Comparative Study and Evaluation

This section presents a detailed comparative study to evaluate the impact of the proposed micro level optimizations on zero-downtime deployments. The study contrasts two distinct deployment environments: a baseline setup representing conventional deployment methodologies and an optimized setup embedding the suite of improvements suggested in this research. The objective is to quantify how these optimizations affect key deployment performance metrics such as downtime, deployment success, recovery time, deployment frequency, and post-deployment stability.

- DNS Swap Technique for BG Deployment

In this technique the DNS routing is changed to point to the Green version when the infrastructure rather than the older Blue version. For this the entire backend infrastructure and the load balancer need to be recreated for the green version. When the traffic needs to be swapped, the DNS router configuration would be changed to the newer version of the infrastructure, so that the newer version of services are delivered to the users.

This technique can be visualized as shown in Fig. 5 below. In here AWS Route 53 DNS is reconfigured to route the traffic to newer ALB. To simulate this pattern, we have used 3 spring boot micro service MS1, MS2 and MS3. These micro services were dockerized and the images were uploaded to Docker hub as I11, I21 and I31. These are the initial version of the services which can be treated as the Blue versions. We used AWS Fargate based ECS (Elastic Container Service) cluster C1 as the hosting mechanism to run the docker containers. We used AWS application load balancer ALB with listeners for the service context paths configured to the target groups which point to the ports exposed for the ECS services.

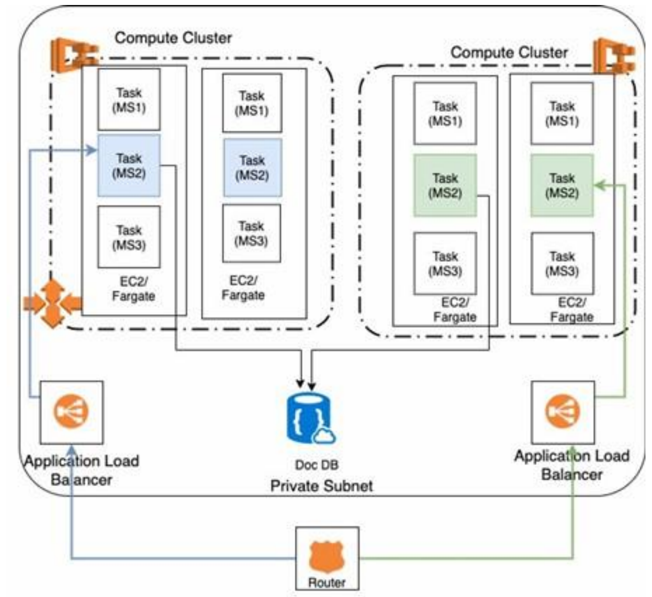


Fig. 5 DNS Swap Technique for Blue Green deployment

We utilized Terraform to configure the AWS infrastructure. After the ECS cluster became operational, we deployed updated versions of the Docker images: I12, I22, and I32, referred to as the Green versions. To transition from the Blue version to the Green versions, we created a duplicate ECS cluster C2, which runs the new microservices MS1, MS2, and MS3 with the updated images I12, I22, and I32 from Docker Hub, paired with the new load balancer ALB2. At this stage, both service versions were operational simultaneously. We then modified the DNS settings to redirect traffic to the new ALB, which directs to the newer ECS cluster C2 through the new load balancer ALB2. We noticed that traffic gradually transitioned from the Blue version to the Green version over time, as it takes a while for DNS changes to propagate. The switching time (SWT) comprises the startup time for the new service (STT), the DNS time to live (TTL), and the ALB health check time interval necessary for routing traffic (HCT).

$WT=STT+TTL+HCT$

WT = Swap Time

STT = Startup Time for the new service

TTL = DNS Time-To-Live (how long cached entries persist)

HCT = Health Check Time (interval before routing traffic to the new instance)

SWT= Switching time

- Load Balancer Swap Technique for BG Deployment

In this technique the DNS routing is kept constant, but the load balancer is updated to point to a newer version of services (green version). This can be visualized using the Fig 5 given below. To simulate this pattern, we have used the same 3 spring boot micro service MS1, MS2 and MS3 and the same setup as described in the earlier section.

When the Blue version of services I11, I21, I31 are up and running, a newer version of services I22, I22, I32 are stood up on a newer version of ECS cluster C2. But in this case, rather than the DNS swap, ALB configuration is changed to route to the newer Green version of the services. This technique needs configurations to adjust the scale up and scale down rules at the ALB level such that the Blue tasks are scaled down and Green tasks are scaled up gradually. The total swap time (SWT2) in this case is a sum of sum of startup time for the new service (STT2) and the ALB health check time interval to start routing the traffic (HCT2).

$SWT_2=STT_2+HCT_2$

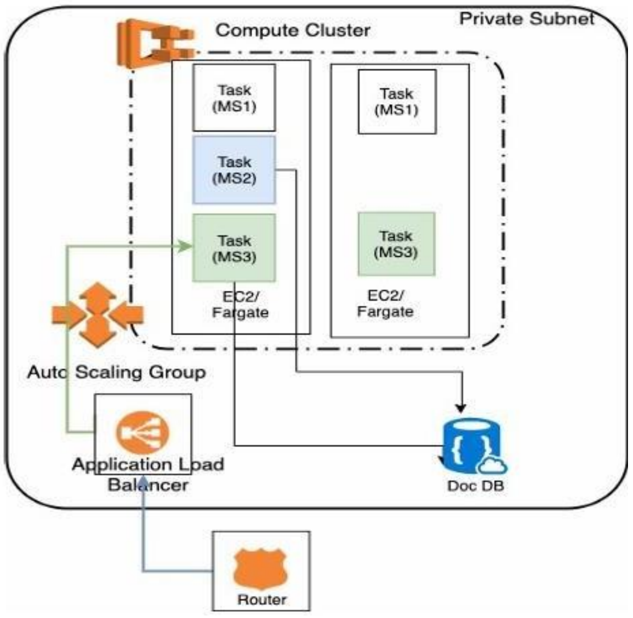


Fig. 9 Load Balancer Swap Technique for Blue Green deployment

VI. LIMITATION AND FUTURE SCOPE

While the research presents an effective framework for achieving zero-downtime deployment, several limitations must be acknowledged. These limitations, along with opportunities for future work, are outlined below:

Limitations

- Environment-Constraint:
  The experiments were conducted exclusively in Kubernetes-based containerized environments. The effectiveness of the proposed techniques in other infrastructure models—such as hybrid cloud, multi-cloud, or legacy on-premises systems—remains unverified.

- Simulated-Traffic:
  The test workloads were synthetic and designed to mimic real-world traffic. However, actual user behavior, burst traffic, session variance, and geographical latency could produce unforeseen issues that were not captured in controlled test environments.

- Stateful Services and Database Complexity: Although Liquibase was used for schema versioning, managing stateful services and highly transactional systems during schema upgrades remains a challenge. Ensuring backward compatibility and live data synchronization across schema versions may require more robust tooling and manual strategies.

- Tool Dependency and Standardization: The solution stack (e.g., Istio, Liquibase, Kubernetes) is not standardized across all enterprises. Integrating this framework into varied toolchains may lead to compatibility issues or necessitate custom development.

Future Scope

- Chaos-Engineering-Integration:
  Incorporating chaos engineering practices could validate the resilience of zero-downtime deployments. Fault injection tools can simulate real-world failures (e.g., pod crashes, DNS failures, latency spikes) to test system behavior during live updates.
- Extension to Serverless and Edge Architectures: As modern applications move toward serverless and edge computing models, adapting zero-downtime deployment principles to these environments will be critical. This includes dynamic scaling, stateless function deployments, and managing updates in distributed edge nodes.

- Cross-Cloud Deployment Evaluation: Future research should evaluate the effectiveness of the proposed framework across multiple cloud providers (AWS, Azure, GCP) to understand platform-specific limitations and optimizations.

VII. Conclusion

This research confirms that achieving zero-downtime deployments in cloud-native environments requires a holistic strategy that integrates architectural best practices, refined deployment techniques, and operational discipline. Techniques like Blue-Green, Rolling, and Canary deployments offer unique advantages depending on application needs, traffic profiles, and infrastructure maturity.

The DNS swap method for Blue-Green deployments, while simple, was found to be inefficient due to its dependence on duplicated infrastructure and high rollback latency tied to DNS TTL settings. Load balancer-based swaps improved performance but required frequent configuration updates. Application-layer swap services offered reduced operational complexity but lacked native support for granular rollout models like canary deployment.

Among these, canary deployment emerged as a particularly effective strategy for balancing risk and agility. Using Kubernetes Gateway API and Istio, this paper demonstrated how traffic splitting and observability tools can be leveraged to gradually expose new versions while preserving service continuity. Additionally, integrating tools like Liquibase enables safe database schema changes without forcing system downtime, addressing a common blind spot in many deployment strategies.

The proposed approach emphasizes micro-level optimizations such as health probes, feature flagging, connection draining, and automated rollback mechanisms. The comparative analysis showed that incorporating these elements significantly reduces deployment-related failures, improves system recovery time, and enhances overall user experience. While implementing such strategies introduces additional complexity, the operational and business benefits outweigh the costs, especially in high-availability environments.

This paper also acknowledges existing limitations in automation, security integration, and multi-cloud compatibility. Future research can explore AI-driven deployment orchestration, chaos engineering, and secure schema evolution for microservice architectures.

In summary, strategic deployment planning—complemented by service mesh technologies, schema management tools, and automated rollback logic—can enable enterprises to confidently adopt zero-downtime practices, achieving both technical resilience and business continuity in software delivery.

References

[1] Chen L. Continuous delivery: Huge benefits, but challenges too. IEEE Software. 2015 Mar;32(2):50-4.

[2] Martin Fowler, "BlueGreenDeployment", 2010, Internet URL: https://martinfowler.com/bliki/BlueGreenDeployment.html

[3] 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," 2018 IEEE International Conference on Services Computing (SCC), San Francisco, CA, 2018, pp. 185-192, doi: 10.1109/SCC.2018.00031.

[4] Lu, Hsin-Ke & Lin, Peng-Chun & Huang, Pin-Chia & Yuan, An. (2017). Deployment and Evaluation of a Continues Integration Process in Agile Development. Journal of Advances in Information Technology. 8. 203 209. 10.12720/jait.8.4.203-209.

[5] Ochei, L., Petrovski, A. & Bass, J. Optimal deployment of components of cloud-hosted application for guaranteeing multitenancy isolation. J Cloud Comp 8, 1 (2019). https://doi.org/10.1186/s13677-018-0124-5

[6] 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

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

[10] S.Hardikar,P.Ahirwar,andS.Rajan,''Containerization:Cloudcomputing based inspiration technology for adoption through Docker and kuber netes,'' in Proc. 2nd Int. Conf. Electron. Sustain. Commun. Syst. (ICESC

[11] A. Malhotra, A. Elsayed, R. Torres, and S. Venkatraman, ''Evaluate solu tions for achieving high availability or near zero downtime for cloud native enterprise applications,'' IEEE Access,

[12] J.DomingusandJ.Arundel,CloudNativeDevopsWithKubernetes:Build ing, Deploying, and Scaling Modern Applications in the Cloud. Beijing, China: O'Reilly, 2022.

[13] R .Jeyaraj, A.Balasubramaniam,N.Guizani,andA.Paul,''Resourceman agement 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.

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

[15] Wang, Y., & Gupta, A. (2023). Multi-Cloud Strategies for Zero-Downtime Deployments: Challenges and Solutions. Cloud Computing Journal, 9(4), 210-225.