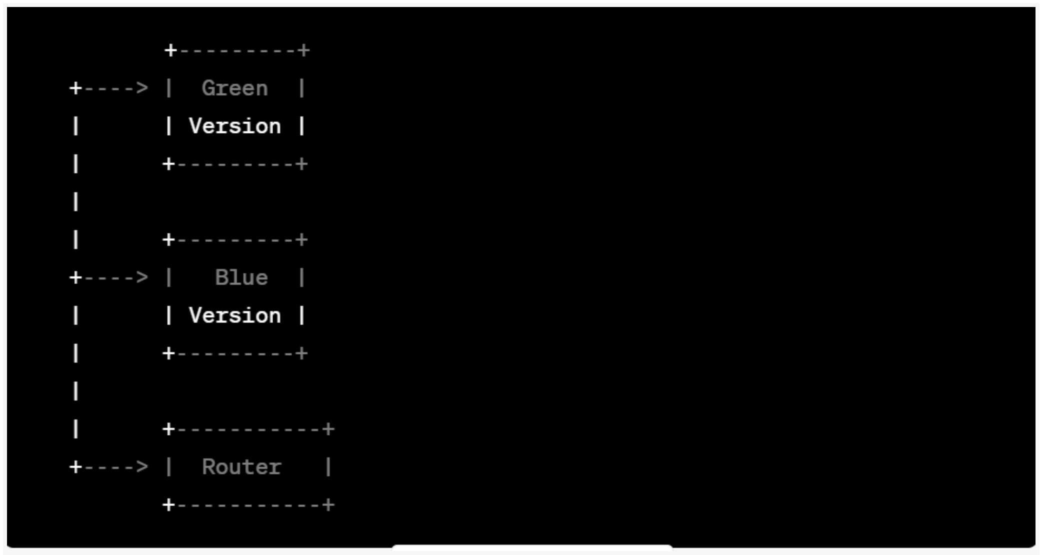
Q1	(a) Being a project manager you are given the task of testing and deploy a new Version (V2) of a mission critical software. The existing version (V1) is now running live. Define a deployment strategy to effectively migrate the current live version with the new version with minimal risk and downtime. Draw a simple schematic of the deployment strategy with various servers and routers involved in it. (Answer your question in terms of Blue/Green Deployments) (b) A leading Europe healthcare player with a large IT structure wanted to adopt agile and DevOps practices to gain true business value. They want to reduce the release cycle time from 12 weeks to 3-4 weeks. Currently they use Java based technologies for waterfall SDLC development of their monolithic applications and they plan to move into cloud. Their team size is 26 where their efforts are distributed among development , manual testing and operations. Please recommend a suitable deployment pipeline that brings business value in terms of faster feature releases, better service quality, efficient deployments with scale and cost optimization. While you recommend a deployment pipeline please write the suitable tools for each stage including continuous monitoring. Explain the benefits
A1 a)	As a project manager, when tasked with testing and deploying a new version (V2) of a mission-critical software while the existing version (V1) is running live, one effective deployment strategy is to utilize the Blue/Green deployment approach. This approach involves setting up a parallel environment (Blue) alongside the production environment (Green), allowing for seamless migration and minimal risk and downtime. Here's a description of the deployment strategy along with a simple schematic: 1. Setup: ● Green Environment: This represents the existing live version (V1) running on production servers. ● Blue Environment: This represents the new version (V2) deployed on separate servers. 2. Testing: ● The new version (V2) is thoroughly tested in the Blue Environment, ensuring it meets all functional and non-functional requirements. 3. Routing Configuration: ● A router or load balancer is set up to handle traffic routing between the Green and Blue environments. This router can be configured to distribute traffic based on specific rules or percentages.

4. Traffic Routing:
    - Initially, all incoming traffic is routed to the Green Environment (V1) where the live version is running. Users continue to interact with V1 without any interruption.
5. Deployment:
    - Once the testing phase is complete, and the new version (V2) is deemed ready for production, the router configuration is updated to start routing a small portion of traffic to the Blue Environment.
6. Gradual Rollout:
    - Traffic is gradually shifted from the Green Environment (V1) to the Blue Environment (V2) in a controlled manner. For example, you could start with routing 10% of the traffic to V2 and monitor its performance.
7. Monitoring and Validation:
    - Continuous monitoring is performed on the Blue Environment to ensure the new version (V2) is functioning correctly and meeting performance expectations. Metrics like response time, error rates, and resource utilization are closely monitored.
8. Full Migration:
    - If the Blue Environment (V2) is stable and performing well, the router configuration is updated to route all traffic to the new version, effectively migrating the system to V2.
9. Rollback Plan:
    - In case any issues or critical failures are detected in the Blue Environment (V2), a rollback plan should be prepared to revert the traffic back to the Green Environment (V1) until the issues are resolved.

Here's a simple schematic representation of the deployment strategy:



```
       +---------+
+----> |  Green  |
|      | Version |
|      +---------+
|
|      +---------+
+----> |   Blue  |
|      | Version |
|      +---------+
|
|      +----------+
+----> |  Router  |
       +----------+
```

In the above diagram, the router is responsible for routing traffic between the Green and Blue environments based on the defined rules or percentages. Initially, all traffic is directed to the Green Environment (V1), and as the rollout progresses, traffic is gradually shifted to the Blue Environment (V2) until it becomes the primary environment.

A1 b)

To enable the leading European healthcare player to adopt agile and DevOps practices, streamline their release cycle, and gain true business value, I recommend implementing a suitable deployment pipeline. Here's a suggested deployment pipeline along with the corresponding tools for each stage:

1. Source Code Management:
    - Tool: Git (e.g., GitLab, GitHub)
    - Benefits: Version control, collaboration, and traceability of source code changes. It allows multiple developers to work on different features concurrently, ensuring code integrity and easy code review.
2. Continuous Integration (CI):
    - Tool: Jenkins, GitLab CI/CD, or CircleCI
    - Benefits: Automates the build and integration process. Developers commit code changes to the source code

repository, triggering automatic builds, code compilation, unit testing, and artifact generation. Early detection of integration issues helps maintain code quality and reduces the risk of conflicts during the later stages.

3. Continuous Deployment (CD) and Deployment Automation:
    - Tool: Kubernetes or Docker Swarm for container orchestration, along with infrastructure-as-code tools like Terraform or AWS CloudFormation for provisioning infrastructure.
    - Benefits: Automates the deployment process, allowing for consistent and reproducible deployments across different environments (e.g., development, staging, production). Containerization enables better scalability, resource utilization, and simplifies application deployment in the cloud environment.
4. Automated Testing:
    - Tools: Selenium or Cypress for UI testing, JUnit or TestNG for unit testing, and tools like JMeter or Gatling for performance testing.
    - Benefits: Automated testing ensures faster feedback loops, reduces manual effort, and improves test coverage. Automated unit tests, integration tests, and UI tests provide confidence in the application's stability and functionality.
5. Continuous Monitoring:
    - Tools: Prometheus for monitoring, Grafana for visualization, and ELK Stack (Elasticsearch, Logstash, and Kibana) for log analysis.
    - Benefits: Monitoring provides real-time visibility into the application's performance, availability, and resource utilization. It helps identify bottlenecks, performance issues, and provides insights for optimizing the system. Log analysis allows quick identification and resolution of issues during runtime.
6. Infrastructure Monitoring:
    - Tools: Infrastructure monitoring tools like Datadog, New Relic, or Nagios.
    - Benefits: Monitors infrastructure components such as servers, databases, and networking devices to ensure their availability, performance, and capacity. It helps identify potential issues and optimize resource utilization.
7. Security Scanning:
    - Tools: OWASP ZAP or SonarQube for security scanning, and Dependency Check for identifying vulnerable

	dependencies. ● Benefits: Scanning for security vulnerabilities helps identify and address potential risks early in the development process. It ensures the application adheres to security best practices and reduces the chances of security breaches. By implementing this deployment pipeline and associated tools, the European healthcare player can achieve the following benefits: ● Faster Feature Releases: The automation and streamlined processes enable faster delivery of new features, reducing the release cycle time from 12 weeks to 3-4 weeks. ● Improved Service Quality: Continuous integration and automated testing help detect issues early in the development cycle, resulting in higher code quality and fewer bugs in production. ● Efficient Deployments with Scale: Containerization and infrastructure-as-code allow for efficient and scalable deployments across different environments, ensuring consistent behavior and reducing the risk of configuration drift. ● Cost Optimization: Cloud adoption and infrastructure-as-code enable the healthcare player to optimize resource allocation, scale resources as needed, and avoid over-provisioning, leading to cost savings. It's important to note that the specific tools and technologies mentioned here are just recommendations, and the healthcare player should assess their unique requirements and choose the tools that best fit their needs and ecosystem.
Q2	What is virtualization? How is Containerization different from Virtualization ? Discuss Docker, Kubernetes and AWS lambda and its suitability to the DevOps implementation. Explain each of these with an example.
A2.	Virtualization: 1. Virtualization is the process of creating a virtual version of an operating system, server, storage device, or network resource. It allows multiple virtual instances to run on a single physical machine, enabling efficient resource utilization and isolation. Virtualization abstracts the underlying hardware, providing flexibility, scalability, and better utilization of computing resources. Each virtual instance behaves as if it were a separate physical entity, running its own operating system and

applications.

Example: Suppose a company needs to run multiple servers for different applications. Instead of purchasing and managing separate physical servers for each application, virtualization allows them to create multiple virtual machines (VMs) on a single physical server. Each VM can have its own operating system and applications, providing isolation and flexibility while maximizing resource utilization.

Containerization:

2. Containerization is a lightweight form of virtualization that isolates applications and their dependencies into containers. Containers package the application and all its required dependencies, libraries, and configuration files into a single portable unit. Unlike traditional virtualization, which emulates an entire operating system, containerization shares the host system's operating system kernel, making containers lightweight, portable, and faster to start and stop.

Example: Suppose a company wants to deploy a microservices-based application. Using containerization, they can create separate containers for each microservice, encapsulating the application code, dependencies, and configurations. Each container can run independently, providing isolation and scalability. Containers can be easily deployed and managed across different environments, ensuring consistency and reducing the chances of compatibility issues.

Docker:

3. Docker is an open-source containerization platform that allows developers to build, package, and distribute applications as containers. It provides a standardized way to create and manage containers, making it easier to deploy and scale applications. Docker enables developers to define application dependencies and configurations using Dockerfiles, which are used to build Docker images. These images can be deployed and run on any system that has Docker installed.

Example: A development team uses Docker to containerize their application. They define a Dockerfile that specifies the application code, dependencies, and configurations. Using Docker, they build an image that encapsulates the application. This image can then be distributed to different environments, such as development, testing, and production, ensuring consistency and reducing deployment issues.

Kubernetes:

4. Kubernetes is an open-source container orchestration platform that automates the deployment, scaling, and management of containerized applications. It provides a highly available and scalable infrastructure for running containers across multiple hosts. Kubernetes manages the scheduling, networking, storage, and scaling of containers, allowing organizations to deploy and manage containerized applications effectively.

Example: A company deploys its containerized application using Kubernetes. They define a Kubernetes manifest file, which describes the desired state of the application, including the number of replicas, resource requirements, networking rules, and other configurations. Kubernetes automatically schedules the containers, manages their lifecycle, and ensures high availability and scalability. If a container fails or the demand increases, Kubernetes automatically restarts failed containers or scales the application by adding more replicas.

AWS Lambda:

5. AWS Lambda is a serverless computing service provided by Amazon Web Services (AWS). It allows developers to run their code without provisioning or managing servers. With AWS Lambda, developers can write functions in various supported languages and trigger them in response to events, such as HTTP requests, file uploads, or database updates. Lambda automatically scales the execution environment based on demand, and users are billed only for the compute time consumed by their functions.

Example: A company wants to process user-uploaded images. They use AWS Lambda to create a serverless function that automatically resizes and compresses the images when they are uploaded to an S3 bucket. Whenever a new image is uploaded, Lambda triggers the function, processes the image, and stores it in another S3 bucket. The company only pays for the compute time used by the function, without worrying about managing servers or infrastructure.

	Suitability to DevOps implementation: ● Docker and Kubernetes: Docker and Kubernetes are popular tools in DevOps practices as they provide consistent environments, scalability, and automation for deploying and managing applications. They enable the development and operations teams to work together seamlessly by facilitating the creation, deployment, and scaling of applications in a reliable and reproducible manner. ● AWS Lambda: AWS Lambda, as a serverless computing platform, allows for event-driven and scalable execution of code without worrying about server provisioning. It can be used in DevOps workflows for automating tasks, implementing serverless architectures, and integrating with other AWS services. Lambda functions can be triggered by various events, enabling continuous integration and deployment pipelines. Overall, Docker, Kubernetes, and AWS Lambda provide valuable tools and technologies for DevOps implementations, promoting collaboration, automation, scalability, and efficient resource utilization in application development and deployment processes.
Q3	(A) A major bank has been observing lots of incidents of late. On close analysis they found that sometime there is performance issue on VMs and they are not receiving any alerts for the same. Also, there is no dashboard for application metrics e.g user login should happen in 5 seconds, payments should be processed in 2 seconds. As there is no application metrics specific dashboard by the time issue impacts the end user it's too late. Based on the information provided please answer the following - (a) Will you recommend Top-down monitoring approach here, or Bottom-Up monitoring approach ? and why ? (b) Recommend some application monitoring tool which could be considered. Application is written in Java (c) As application is deployed on AWS EC2 AutoScalingGroup, so scaling up and scaling down is a norm, so how we can ensure that new instances coming up because of scaling has entire monitoring enabled. (B) ABC organization would like to monitor the following scenarios occurring at their end continuously

	(a) Suspicious Login Attempts (b) Network Failure or downtime (c) User sessions buffer cache (d) Tracking VMs 1 mark ABC Organization must apply Application Monitoring, Network Monitoring, Database Monitoring or Security Monitoring for each of these A scenarios. Select the correct monitoring type and explain your choice
A.3 a)	(a) In the given scenario, I would recommend a Bottom-Up monitoring approach. The Bottom-Up monitoring approach involves monitoring individual components and services of the system to identify any performance issues or anomalies. Since the bank is experiencing performance issues on VMs and not receiving alerts, it indicates a problem at the infrastructure level. By implementing Bottom-Up monitoring, the bank can proactively monitor the VMs and detect any performance issues or failures before they impact the end users. (b) For application monitoring in Java, I recommend using Prometheus in combination with Grafana. Prometheus is an open-source monitoring and alerting toolkit that is well-suited for monitoring Java applications. It has strong support for Java application monitoring through client libraries such as the Prometheus Java client. Prometheus allows you to collect and store application metrics and provides powerful querying and alerting capabilities. Grafana, on the other hand, is a popular open-source data visualization and monitoring tool. It can be integrated with Prometheus to create interactive and

	customizable dashboards for monitoring application metrics. Grafana provides various visualization options, alerting features, and the ability to create dynamic and informative dashboards. (c) To ensure that new instances coming up due to scaling have monitoring enabled, you can leverage AWS CloudFormation or Infrastructure as Code (IaC) tools such as AWS Cloud Development Kit (CDK) or Terraform. By using CloudFormation or IaC tools, you can define the infrastructure configuration including monitoring settings, such as enabling monitoring agents or installing monitoring tools, in a template or script. When new instances are launched as part of the Auto Scaling Group, the defined monitoring configuration will be automatically applied to the new instances, ensuring that they have monitoring enabled from the start.
A.3 b)	For the scenarios mentioned by ABC organization, here are the recommended monitoring types: (a) Suspicious Login Attempts: Security Monitoring Security Monitoring focuses on detecting and responding to security-related events and threats. To monitor suspicious login attempts, ABC organization should implement Security Monitoring. This can be done through intrusion detection systems (IDS) or security information and event management (SIEM) tools. These tools analyze network traffic, log data, and authentication events to identify potential security breaches or unauthorized access attempts. By continuously monitoring and analyzing login attempts, ABC organization can detect and respond to any suspicious activity, protecting their systems and sensitive data. (b) Network Failure or Downtime: Network Monitoring Network Monitoring involves monitoring the performance and availability of network infrastructure components. To monitor network failures or downtime, ABC organization should implement Network

	Monitoring. Network monitoring tools such as Nagios, Zabbix, or PRTG can be used. These tools provide real-time visibility into network devices, connections, and bandwidth usage. By continuously monitoring the network, ABC organization can quickly identify and resolve any network failures or downtime, minimizing the impact on their operations. (c) User Sessions, Buffer Cache: Application Monitoring Application Monitoring focuses on monitoring the performance and behavior of applications. To monitor user sessions and buffer cache, ABC organization should implement Application Monitoring. Application performance monitoring (APM) tools such as New Relic or Dynatrace can be used. These tools provide insights into application performance metrics, user sessions, and cache utilization. By continuously monitoring application metrics, ABC organization can identify potential performance bottlenecks, optimize their applications, and ensure a smooth user experience. (d) Tracking VMs: Infrastructure Monitoring Infrastructure Monitoring involves monitoring the performance and availability of infrastructure components such as servers, virtual machines (VMs), and storage. To track VMs, ABC organization should implement Infrastructure Monitoring. Infrastructure monitoring tools like Datadog or Prometheus can be utilized. These tools provide visibility into VM resource utilization, performance metrics, and health status. By continuously monitoring VMs, ABC organization can track their performance, identify any resource constraints, and ensure the scalability and availability of their infrastructure. In summary: ● Suspicious Login Attempts: Security Monitoring ● Network Failure or Downtime: Network Monitoring ● User Sessions, Buffer Cache: Application Monitoring ● Tracking VMs: Infrastructure Monitoring
Q4	(a) Pfizer have their Data Centre (Production Centre) on Google Public Cloud in Ohio State in USA. The data Centre is built with proper Network resources VPC, Sub-nets, load balancers. Fire walls etc to deploy the Pfizer e-commerce application on to the Ohio Environment. Pfizer also wanted to have a Disaster Recovery Centre in another Region (Seattle) but they wanted to have Data Centre and Disaster Recovery Centre in Active-Passive mode. How do you build DR Environment on demand with Infra as Code using Terraform? Pl lay

	down the Architecture with all the steps for the same (b) If resources are locked into a particular configuration which does have preset values for CPU, Memory, Networking that does not expand as demand grows and does not shrink when there is no demand. Then what kind of mechanism will you use on Cloud. Explain with an example. Also how do you control the economics of the resources on the cloud when you adopt to the specified mechanism.
A4 - a)	To build a Disaster Recovery (DR) environment for Pfizer's e-commerce application on Google Public Cloud using Terraform, you can follow the steps outlined below. This assumes you have Terraform installed and have knowledge of working with Google Cloud Platform (GCP). 1. Set up the Terraform project: ● Create a new directory for your Terraform project. ● Initialize the project by running the command: `terraform init`. 2. Define provider and authentication: ● Create a `provider.tf` file and define the GCP provider and authentication details. ● Specify the project, credentials, and any other necessary configuration. 3. Define the infrastructure components: ● Create a new Terraform module for the DR environment. ● Define the required infrastructure components such as VPC, subnets, load balancers, firewalls, etc., in separate Terraform files. ● Organize the files and modules according to your preferred structure. 4. Define the active-passive DR architecture: ● Determine the desired architecture for the active-passive DR setup. ● Identify the resources that need to be replicated in the DR region

    (Seattle) to achieve the active-passive configuration.
    - For example, you may need to replicate the VPC, subnets, load balancers, storage, and relevant networking configurations.
5. Configure the replication and failover mechanisms:
    - Set up the replication mechanism for critical data and resources between the primary data center (Ohio) and the DR center (Seattle).
    - Use GCP services like Cloud Storage, Cloud SQL, or any other relevant services for data replication.
    - Configure the failover mechanism to redirect traffic to the DR environment when the primary data center is unavailable.
    - This could involve updating DNS records, modifying load balancer configurations, or using traffic management services like Traffic Director.
6. Write the Terraform code:
    - Use Terraform's GCP provider to define the necessary resources and configurations for the DR environment.
    - Create Terraform files that describe the infrastructure components, resource dependencies, and any other required configurations.
    - Leverage Terraform modules, variables, and data sources to organize and reuse your code effectively.
7. Plan and apply the Terraform configuration:
    - Run `terraform plan` to see the execution plan and verify that the infrastructure matches your expectations.
    - Review the plan carefully to ensure all the desired components and configurations are included.
    - Run `terraform apply` to apply the configuration and provision the DR environment on Google Public Cloud.
8. Test and validate the DR setup:

	● Perform thorough testing to ensure the active-passive DR setup functions as expected. ● Conduct tests for failover scenarios and verify that the application and its components are available in the DR environment. ● Validate that data replication and synchronization mechanisms are working correctly. By following these steps, you can leverage Terraform to provision the necessary infrastructure and configurations for an active-passive Disaster Recovery environment for Pfizer's e-commerce application on Google Public Cloud. Remember to regularly update and test your DR environment to ensure its effectiveness in case of a disaster.
A4 - b)	When resources have locked configurations that do not automatically expand or shrink based on demand, you can use Auto Scaling Groups (ASGs) or similar mechanisms provided by the cloud platform. Auto Scaling Groups allow you to automatically adjust the number of instances (virtual machines) based on demand. Here's an example using AWS Auto Scaling: 1. Define an Auto Scaling Group (ASG) with the desired locked configuration, including CPU, memory, and networking specifications. 2. Set up a scaling policy based on predefined metrics such as CPU utilization or network traffic. 3. Configure the ASG to automatically scale out (increase the number of instances) when the demand exceeds a certain threshold. 4. Similarly, configure the ASG to scale in (decrease the number of instances) when the demand decreases below a certain threshold.

With this mechanism, the resources will automatically scale up or down based on demand, ensuring optimal utilization and cost efficiency. For example, during peak times, the ASG will automatically add more instances to handle increased traffic, and during periods of low demand, it will reduce the number of instances to save costs.

To control the economics of resources on the cloud when using such mechanisms, you can set up policies and guidelines:

1. Utilize cost optimization tools provided by the cloud platform to analyze resource usage and identify areas for optimization.
2. Set up budget alerts and notifications to monitor and manage your cloud spending.
3. Use reserved instances or savings plans to get discounted pricing for long-term resource commitments.
4. Leverage cloud-native services such as serverless computing (e.g., AWS Lambda) or managed databases to pay only for the actual usage.
5. Regularly review and optimize resource configurations based on usage patterns and business requirements to avoid overprovisioning.

By implementing these mechanisms and cost control strategies, you can ensure efficient resource utilization while managing the economics of cloud resources effectively.