**1.0 Introduction to DevOps**

**1.1 DevOps Principles and Culture**

* **Key Concepts in DevOps**: DevOps is a set of practices that combines software development (Dev) and IT operations (Ops) to shorten the systems development life cycle while delivering features, fixes, and updates frequently in close alignment with business objectives.
* **Importance in Software Development**: It aims to foster a culture of collaboration between development and operations teams, automating processes to achieve efficiency, reliability, and faster delivery of applications and services.

**Lifecycle of DevOps and the Importance of Automation**

* **DevOps Lifecycle**: Includes continuous integration, continuous delivery/deployment, and continuous monitoring, focusing on automation throughout.
* **Importance of Automation**: Automation reduces human error, speeds up processes, and ensures consistency in deploying and managing applications and infrastructure.

**2.0 CI/CD Fundamentals**

**2.1 Building a CI/CD Pipeline and Understanding Its Components**

* **CI/CD Pipeline**: Automates the processes of building, testing, and deploying code. It typically includes stages like code compilation, unit testing, integration testing, and deployment to production.
* **Components**: Includes version control, build server, testing environment, and deployment tools.

**Overview of CI/CD Tools and Comparison**

* **CI Tools**: Jenkins, Travis CI, CircleCI, GitLab CI.
* **CD Tools**: Ansible, Puppet, Chef, Kubernetes for orchestration.

**3.0 Introduction to Docker**

**3.1 Concepts of Containerization**

* **Virtualization vs. Containerization**: Virtualization abstracts hardware to run multiple OS instances; containerization abstracts the OS to run multiple isolated applications.
* **Impact on Development**: Containers provide consistency across different environments, simplify dependencies, and streamline deployment.

**The Rise of Containers and Their Impact on Development**

* **Use Cases**: Microservices architecture, portable development environments, and scaling applications.
* **Advantages**: Lightweight, fast startup, resource efficient, consistent across environments.

**3.2 Docker Ecosystem and Architecture**

* **Components**: Docker daemon (runs on host), Docker client (interacts with daemon), images (read-only template), containers (running instance of image), registries (store and distribute images).
* **Docker Engine**: Combines daemon, REST API, and command-line interface.

**The Role of Docker Hub and Container Registries**

* **Docker Hub**: Official repository for Docker images, public and private registries for storing and sharing images.

**3.3 Working with Docker Containers**

* **Basic Docker Commands**: docker pull, docker run, docker exec, docker stop, docker start, docker rm.
* **Interacting with Containers**: Viewing logs (docker logs), executing commands inside containers (docker exec).
* **Managing Lifecycles**: Creating, starting, stopping, and removing containers.

**3.4 Setting Up Your Docker Environment**

* **Installation**: Docker is available for Windows, macOS, and Linux. Install using respective installers or package managers.
* **Running Your First Container**: Pull an image (docker pull), run a container (docker run), interact with it (docker exec).

**4.0 Docker Compose and Container Management**

**4.1 Building and Managing Docker Images**

* **Writing Dockerfiles**: Instructions to build Docker images, specifying base image, dependencies, environment variables, and commands.
* **Building Images from a Dockerfile**: Use docker build command with -t tag to create an image from a Dockerfile.

**4.2 Multi-container Applications with Docker Compose**

* **Introduction to Docker Compose**: Tool for defining and running multi-container Docker applications.
* **Compose File Structure**: YAML file (docker-compose.yml) defining services, networks, volumes, and dependencies.
* **Orchestrating with Docker Compose**: Simplifies launching interconnected containers, manages container lifecycle.

**4.3 Container Networking and Storage**

* **Docker Networking**:
  + none: Isolated containers, no networking.
  + bridge: Default, containers on the same host can communicate.
  + host: Uses host network stack.
  + overlay: Across multiple Docker daemons, requires Swarm or Kubernetes.
* **Persistent Data and Volumes**: Persistent storage for container data.
* **Volume Sharing**: Between containers (--volumes-from) and between host and containers (-v flag).

**4.4 Creating a Multi-container Application**

* **Developing Docker Compose File**: Define services, volumes, networks, and dependencies.
* **Building and Running**: Use docker-compose up to build, create, start, and attach to containers.
* **Managing State**: Ensure data persistence with volumes, configure networks for inter-container communication.

**5.0 Introduction to Kubernetes**

Kubernetes is an open-source container orchestration platform designed to automate the deployment, scaling, and management of containerized applications. It provides a framework for automating the deployment, scaling, and operations of application containers across clusters of hosts.

**5.1 Kubernetes Core Concepts**

**Pods**

* Pods are the smallest deployable units in Kubernetes.
* A Pod can contain one or more containers that share resources like storage and network.
* Pods represent a logical application that can consist of a single container or a tightly coupled group of containers.

**Services**

* Kubernetes Services enable communication and discovery between Pods.
* They provide a stable endpoint (IP address and port) that remains consistent even if Pods are added or removed.
* Services can be exposed internally within the cluster or externally to the internet.

**Deployments**

* Deployments manage the deployment and scaling of Pods and ReplicaSets (which manage multiple instances of Pods).
* They ensure that a specified number of replica Pods are running at any given time.
* Deployments enable updates to applications by creating a new ReplicaSet and gradually updating Pods.

**5.2 Kubernetes Cluster Architecture**

**Components of a Kubernetes Cluster**

* **Master Node:** Controls the Kubernetes cluster. Components include:
  + **Kubernetes API Server:** Exposes the Kubernetes API, which is used by all other components.
  + **etcd:** Consistent and highly available key-value store used as Kubernetes' backing store for all cluster data.
  + **Scheduler:** Assigns Pods to Nodes based on resource availability.
  + **Controller Manager:** Monitors the state of the cluster and performs routine tasks.
* **Worker Node:** Where application containers (Pods) run. Components include:
  + **kubelet:** Agent that runs on each node and ensures Pods are running and healthy.
  + **kube-proxy:** Manages network communication between Pods and services.

**Other Key Components**

* **kubectl:** Command-line tool used to interact with Kubernetes clusters.
* **Container Runtime:** Software that runs containers (e.g., Docker, containerd).

**5.3 Deploying Applications on Kubernetes**

**Writing YAML Manifests**

* Kubernetes resources are defined in YAML manifests (e.g., Deployment, Service).
* Manifests specify desired state (e.g., number of replicas, container images, ports).

**Deploying and Managing Applications**

* Use kubectl apply to create or update resources defined in YAML manifests.
* kubectl commands can be used to inspect cluster resources, manage deployments, and troubleshoot issues.

**Debugging and Logging**

* Kubernetes provides logging mechanisms to capture container logs.
* Use kubectl logs to retrieve logs from a specific container within a Pod.
* Debugging involves inspecting Pod state, logs, and events (kubectl describe).

**5.4 Running Your First App in Kubernetes**

**Setting Up a Local Environment**

* **Minikube:** Tool to run a single-node Kubernetes cluster locally.
* **Kind (Kubernetes in Docker):** Tool for running Kubernetes clusters using Docker containers as nodes.

**Deploying a Simple Stateless Application**

* Write a Deployment YAML manifest specifying container image, ports, and replicas.
* Apply the manifest using kubectl apply -f deployment.yaml.

**Exposing the Application**

* Define a Service YAML manifest to expose the application.
* Use kubectl apply -f service.yaml to create the Service.

**6.0 Kubernetes Orchestration and Scaling**

**6.1 Advanced Deployment Strategies**

**Rolling updates and rollbacks:**

* **Rolling Updates:** Kubernetes allows for rolling updates to application deployments, meaning that updates are applied to a subset of pods at a time, ensuring minimal downtime and gradual transition.
* **Rollbacks:** In case of issues with a new deployment, Kubernetes supports rollbacks to a previous stable version. This is done automatically if configured, or manually by specifying the previous version.

**Blue-green and canary deployments:**

* **Blue-Green Deployments:** In this strategy, you maintain two identical environments (blue and green). You route production traffic to one environment (say, blue), while deploying updates to the other (green). Once green is validated, traffic is switched to it.
* **Canary Deployments:** Canary deployments involve gradually rolling out updates to a small subset of users or traffic. This allows monitoring for issues before deploying changes widely.

**Use of readiness and liveness probes:**

* **Readiness Probes:** Ensure that pods are ready to serve traffic before sending requests to them. If a pod's readiness probe fails, it is removed from service until it passes.
* **Liveness Probes:** Determine if pods are running correctly. If a liveness probe fails, Kubernetes restarts the container automatically.

**6.2 Service Discovery and Load Balancing**

**Deep dive into Kubernetes Service types:**

* **ClusterIP:** Default type, exposes the service on an internal IP within the cluster.
* **NodePort:** Exposes the service on each Node’s IP at a static port. Accessible from outside the cluster.
* **LoadBalancer:** Creates an external load balancer in cloud providers (AWS, GCP, etc.) and assigns a fixed external IP to the service.
* **ExternalName:** Maps the service to a DNS name outside the cluster.

**Implementing service discovery within a cluster:**

* Kubernetes uses DNS for service discovery. Each service gets its own DNS name, which other pods can use to connect.

**Ingress controllers and Ingress resources for external access:**

* **Ingress Controllers:** Manage external access to services in a cluster, typically HTTP/HTTPS traffic. They route traffic based on rules defined in Ingress resources.
* **Ingress Resources:** Kubernetes resources that define rules and configurations for routing external traffic to services.

**6.3 Scaling Applications in Kubernetes**

**Horizontal Pod Autoscaling based on metrics:**

* Automatically adjusts the number of replica pods in a deployment based on observed CPU or custom metrics utilization.
* Ensures optimal performance without over-provisioning resources.

**Manual vs automatic scaling:**

* **Manual Scaling:** Adjusting the number of replica pods manually based on anticipated load or specific requirements.
* **Automatic Scaling:** Kubernetes handles scaling based on defined metrics and policies, reducing the need for manual intervention.

**6.4 Scaling and Managing Stateful Applications**

**Implementing a Horizontal Pod Autoscaler:**

* Allows automatic scaling of stateless applications based on CPU or custom metrics.

**Managing stateful applications with StatefulSets:**

* Specialized controller for managing stateful applications (e.g., databases) that require stable, unique network identifiers and stable storage.

**Persistent storage solutions with PersistentVolumes and PersistentVolumeClaims:**

* **PersistentVolumes (PV):** Abstract storage from underlying storage systems, allowing cluster administrators to provision storage resources.
* **PersistentVolumeClaims (PVC):** Requests for storage by users. PVCs consume PVs.

**7.0 Helm and Kubernetes Ecosystem**

**7.1 Introduction to Helm**

**Overview of Helm and its advantages for Kubernetes package management:**

* **Helm** is a package manager for Kubernetes that simplifies the process of deploying and managing applications on Kubernetes clusters.
* Advantages:
  + **Reusability:** Helm Charts (packages) encapsulate Kubernetes resources, making them easy to share and reuse.
  + **Templating:** Charts use Go templates, allowing parameterization and customization of Kubernetes manifests.
  + **Versioning:** Helm manages versioning and dependency management for Kubernetes applications.
  + **Rollback and Upgrade:** Facilitates rollback and upgrade operations for Kubernetes applications.

**Understanding Helm Charts, Releases, and Repositories:**

* **Helm Chart:** A collection of Kubernetes YAML manifests (Deployment, Service, ConfigMap, etc.) organized into a directory structure. Charts can be templated to allow for customization.
* **Release:** An instance of a Helm Chart installed into a Kubernetes cluster. Each release has a unique name and can be versioned.
* **Repository:** A collection of Helm Charts that can be searched, installed, and shared. Helm charts are typically stored in repositories (public or private).

**Helm installation and basic commands:**

* **Installation:** Helm can be installed on various platforms (Linux, macOS, Windows). Typically installed using package managers or binaries.
* **Basic Commands:**
  + helm search: Search available Helm charts in repositories.
  + helm install: Install a Helm chart into a Kubernetes cluster.
  + helm upgrade: Upgrade a release to a new version of a Helm chart.
  + helm rollback: Rollback to a previous version of a release.
  + helm list: List installed releases.
  + helm delete: Uninstall a release from the Kubernetes cluster.

**7.2 Managing Kubernetes Resources with Helm**

**Creating and customizing Helm Charts:**

* **Creating Charts:** Use helm create command to scaffold a new Helm chart. Charts consist of templates, values, and metadata files.
* **Customizing Charts:** Edit values.yaml to define default configurations and override values during installation.

**Templating resources and managing releases with Helm:**

* **Templating:** Helm uses Go templating to dynamically generate Kubernetes manifest files. This allows parameterization and reuse of configurations across deployments.
* **Managing Releases:** Each release is managed independently. Helm tracks releases by name, allowing multiple instances of the same chart with different configurations.

**Rollbacks and upgrades with Helm:**

* **Rollbacks:** Helm supports rolling back to a previous release version if an upgrade fails or causes issues. This ensures resilience and reliability in deployments.
* **Upgrades:** Helm facilitates seamless upgrades of deployed applications by managing changes to Kubernetes resources based on updated Helm charts or configurations.

**8.0 Continuous Integration with Jenkins**

**8.1 Setup and Configuration of Jenkins**

**Introduction to Jenkins and Continuous Integration:**

* **Jenkins** is an open-source automation server used for continuous integration and continuous delivery (CI/CD) pipelines.
* **Continuous Integration (CI)** involves automating the build and testing of code whenever changes are committed to version control. Jenkins facilitates this process by automating repetitive tasks.

**Installing and configuring Jenkins:**

* **Installation:** Jenkins can be installed on various operating systems (Linux, macOS, Windows) and cloud platforms (AWS, GCP, Azure).
* **Configuration:** After installation, Jenkins is accessed via a web interface. Initial setup involves configuring security, plugins, and administrative settings.

**Navigating the Jenkins dashboard and setting up a build node:**

* **Dashboard:** Central hub for managing Jenkins jobs, views, and administrative tasks.
* **Build Node:** Jenkins distributes workload to build nodes (agents or slaves) where builds are executed. Nodes can be configured based on OS, tools, and environment requirements.

**8.2 Building CI Pipelines with Jenkins**

**Creating and managing Jenkins jobs:**

* **Jobs:** Jenkins jobs define tasks such as building, testing, and deploying applications. Jobs are configured via the Jenkins web interface.
* **Freestyle vs. Pipeline:** Freestyle jobs allow flexible configuration, while Pipeline jobs (using Jenkinsfile) enable defining pipelines as code for better version control and scalability.

**Understanding Jenkinsfile and pipeline as code:**

* **Jenkinsfile:** Declarative or scripted file defining the entire pipeline flow using Groovy syntax. It defines stages, steps, and conditions for automation.
* **Pipeline as Code:** Advantages include versioning, reproducibility, and easier collaboration among teams.

**Triggering builds on code changes:**

* Jenkins monitors version control systems (e.g., Git, SVN) for changes. Builds can be triggered automatically (polling SCM) or manually.

**8.3 Integrating Automated Tests**

**Configuring build steps to include testing frameworks:**

* Jenkins integrates with various testing frameworks (JUnit, Selenium, etc.) via plugins. Tests are executed as part of the build process to validate code changes.

**Reporting test results and managing test artifacts:**

* Test results are captured and displayed in Jenkins. Plugins like JUnit and TestNG provide detailed reports.
* Artifacts (compiled code, test reports) are archived for traceability and future reference.

**Setting up notifications for build outcomes:**

* Jenkins notifies stakeholders (developers, QA, etc.) about build outcomes via email, chat (Slack, Microsoft Teams), or custom notifications.
* Notifications help in monitoring build status and addressing issues promptly.

**8.4 Building and Running a Jenkins Pipeline**

**Writing a Jenkinsfile for a multi-branch pipeline:**

* Multi-branch pipelines manage different branches (e.g., feature, development) in version control, automatically triggering builds based on changes.

**Setting up a pipeline for a Java application with automated tests:**

* Example scenario involves building a Java application, running unit tests (JUnit), and packaging artifacts (JAR/WAR).

**Triggering the pipeline through a simulated code change and reviewing the output:**

* Simulate a code change (commit/push) to the version control system.
* Jenkins detects the change, triggers the pipeline defined in Jenkinsfile, executes stages (build, test, deploy), and displays output/logs in the Jenkins UI.

**9.0 Continuous Deployment (CD) with Jenkins**

**9.1 Implementing Continuous Deployment with Jenkins**

**The principles of Continuous Deployment:**

* **Continuous Deployment (CD)** automates the release process to deliver code changes to production environments seamlessly and frequently.
* Principles include automated testing, continuous integration, and deployment automation to reduce manual intervention and speed up delivery.

**Extending Jenkins pipelines for CD:**

* **Pipeline stages:** Expand Jenkins pipelines (defined in Jenkinsfile) to include deployment stages after successful build and testing phases.
* **Deployment strategies:** Define deployment steps, such as deploying to staging environments first, before promoting to production.

**Approaches to safe deployments: feature flags, A/B testing, canary releases:**

* **Feature Flags:** Toggle new features on/off in production to manage risk and gather user feedback.
* **A/B Testing:** Serve different versions of an application to different user segments to compare performance and user response.
* **Canary Releases:** Gradual rollout of new features or versions to a subset of users to validate stability before full deployment.

**9.2 Jenkins Integration with Containerized Workflows**

**Building Docker images with Jenkins:**

* Use Jenkins pipelines to automate Docker image builds from Dockerfiles or existing source code repositories.
* Jenkins plugins (e.g., Docker Pipeline Plugin) facilitate building and tagging Docker images.

**Pushing images to a registry using Jenkins:**

* Jenkins pipelines can automate the process of pushing built Docker images to Docker registries (e.g., Docker Hub, AWS ECR, Google Container Registry).

**Deploying to Kubernetes from Jenkins:**

* Jenkins integrates with Kubernetes clusters to automate deployment of Dockerized applications.
* Use Kubernetes CLI (kubectl) or Jenkins Kubernetes plugin to interact with Kubernetes API for deployments.

**9.3 Implementing a CD Pipeline for a Containerized Application**

**Updating the Jenkinsfile to include deployment to a Kubernetes environment:**

* Extend Jenkinsfile to include deployment stages after build and test stages. Example stages: Build Docker image, Push image to registry, Deploy to Kubernetes.

**Configuring Jenkins with credentials for Docker registry and Kubernetes cluster access:**

* Store Docker registry credentials (username, password) and Kubernetes cluster credentials securely in Jenkins.
* Use Jenkins Credentials Plugin to manage sensitive information.

**Executing the full CI/CD pipeline and verifying the deployment on Kubernetes:**

* Trigger the Jenkins pipeline manually or automatically on code changes.
* Jenkins executes the pipeline: builds Docker image, pushes to registry, deploys to Kubernetes.
* Monitor Jenkins console output and Kubernetes dashboard for deployment status and verify application functionality in Kubernetes environment.