EE 6254: DEVOPS ENGINEERING

ASSIGNMENT 3

Group Number : 40

Group Members : EG/2020/4068 – Manahara U.V.S.

EG/2020/4321 – Soysa W.S.N

Date : 27/06/2024

**Part 1: CICD Design Diagram**

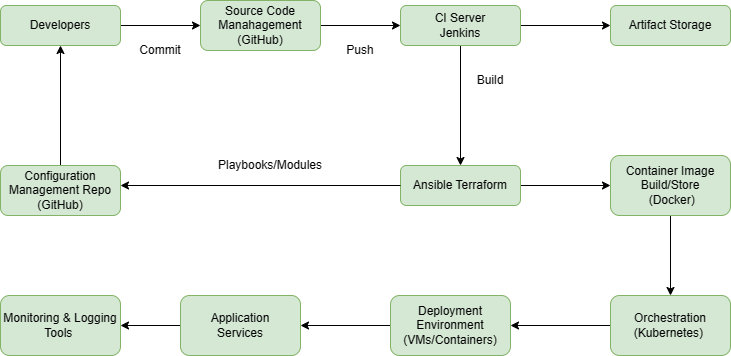


Figure 1: CICD Design Diagram

**Explanation**

**Developers**

Developers are responsible for writing and maintaining the application's code. They use various programming languages and tools to build new features, fix bugs, and improve performance. After making changes, developers commit their code to a Git repository, such as GitHub. This process involves creating a commit message that describes the changes and pushing the code to a central repository. Version control systems like Git allow developers to track changes, collaborate with team members, and revert to previous versions if needed. Code reviews are often conducted to ensure code quality and adherence to coding standards.

**Git Repository**

A Git repository serves as the central storage for the codebase, providing version control and efficient management of the code. It tracks every change made to the codebase, allowing developers to revert to previous versions if necessary. Branching and merging are key features of Git, enabling multiple developers to work on different features or fixes simultaneously without interfering with each other’s work. Pull requests facilitate code reviews and discussions before changes are merged into the main branch. The repository also integrates with CI/CD pipelines to automate the build and deployment processes.

**CI Server (Jenkins)**

Jenkins is the core tool used for Continuous Integration (CI). It automates the process of building, testing, and integrating code changes to ensure that the code is always in a deployable state. Whenever a commit or push is detected in the Git repository, Jenkins pulls the latest code, builds it, and runs a series of tests to validate its functionality. These tests can include unit tests, integration tests, and end-to-end tests. Jenkins then packages the application into a deployable format, such as a Docker image or a JAR file. It also provides detailed logs and reports, helping developers quickly identify and fix issues.

**Artifact Storage**

Once Jenkins successfully builds the application, it pushes the build artifacts to an artifact storage repository, such as Nexus or Artifactory. These repositories are designed to store, version, and manage build artifacts, ensuring that the exact version of the application that was tested and built can be deployed. Artifact storage repositories provide a central location for all build artifacts, making them easily accessible for deployment, testing, and rollback. They also support various types of artifacts, including Docker images, JAR files, and npm packages. Security features like access control and vulnerability scanning help protect the artifacts.

**Configuration Management and IaC Tools (Ansible/Terraform)**

For managing and provisioning our infrastructure and application configurations, we use tools like Ansible and Terraform. Ansible handles configuration management, ensuring that all systems are configured correctly and consistently. It uses a declarative language to define configurations and can automate complex tasks, such as installing software, managing users, and configuring services. Terraform is used for Infrastructure as Code (IaC), allowing us to define and provision infrastructure, such as virtual machines (VMs), networks, and storage, in a declarative manner. It supports multiple cloud providers and allows for versioning of infrastructure changes, enabling teams to track and roll back changes as needed.

**Container Image Build/Store (Docker)**

Our application is containerized using Docker, which allows us to package the application and its dependencies into a single, portable container. Docker images are built using Dockerfiles, which specify the application’s environment, dependencies, and startup commands. Once built, these images are stored in a Docker registry, such as Docker Hub or a private registry. The Docker registry provides versioning and ensures that the correct images are used during deployment. It also supports image scanning to detect vulnerabilities and enforce security policies. Containerization helps ensure consistency across different environments, from development to production.

**Orchestration (Kubernetes)**

Kubernetes manages container orchestration, handling the deployment, scaling, and operations of application containers. It uses a declarative approach to define the desired state of the application and ensures that the actual state matches the desired state. Kubernetes pulls Docker images from the Docker registry and deploys them within the cluster, ensuring high availability and scalability. It provides features like load balancing, self-healing, and rolling updates to ensure that the application runs smoothly and can handle increased traffic. Kubernetes also supports service discovery and networking, allowing containers to communicate with each other and with external services.

**Deployment Environment (VMs/Containers)**

The deployment environment consists of virtual machines (VMs) or containers where the application runs. VMs provide an isolated environment with a dedicated operating system, while containers share the host OS kernel but provide isolated user space. Kubernetes manages the deployment of containers on these VMs or container hosts, ensuring efficient resource use and seamless updates. It schedules containers based on resource requirements and availability, and it can automatically scale the number of containers up or down based on demand. This ensures that the application can handle varying levels of traffic without manual intervention.

**Application Services**

Application services run within the containers managed by Kubernetes. These services include various components of the application, such as web servers, databases, and background workers. Kubernetes manages the communication between these services using a service mesh, which provides features like traffic management, load balancing, and security. The service mesh ensures that services can communicate with each other securely and efficiently, regardless of where they are deployed within the cluster. It also provides observability features, such as tracing and monitoring, to help identify and troubleshoot issues within the application.

**Monitoring & Logging Tools**

To maintain the health and performance of our application and infrastructure, we use monitoring and logging tools like Prometheus and Grafana. Prometheus collects metrics from various components of the application and infrastructure, such as CPU usage, memory usage, and response times. Grafana visualizes these metrics, providing dashboards and alerts to help us monitor the system’s health. Logging tools collect logs from the application and infrastructure, allowing us to analyze and troubleshoot issues. Logs can be centralized and indexed for easy searching, and they can be integrated with alerting systems to notify us of critical issues.

**Connectivity in the CI/CD Pipeline**

* Developers push code to the Git repository.
* Jenkins pulls the code from the Git repository, builds it, and pushes the build artifacts to the artifact storage.
* Ansible and Terraform pull configurations from the configuration management repository and provision the necessary infrastructure.
* Jenkins triggers Docker image builds, which are stored in a Docker registry.
* Kubernetes pulls these Docker images and deploys them to the deployment environment.
* Application services communicate over the Kubernetes network.
* Monitoring and logging tools collect metrics and logs from the deployment environment and application services to ensure smooth operation.

**Part 2: Automation Approach**

**Application Tools and Dependencies**

**React**

* Version: 17.0
* Purpose: Application framework for building webapplications.

**MongoDB**

* Version: 7.0
* Purpose: Database management system for storing and managing application data.

**Automated Application Deployment**

**DevOps Tools**

**GitHub**

* Version: Latest Stable
* Purpose: GitHub is used for version control and as a code repository. It manages and tracks code changes, enabling collaboration among developers. GitHub supports branching, pull requests, and code reviews, ensuring code quality and smooth integration of new features. It integrates with CI/CD pipelines, triggering automated processes like builds and deployments upon code commits.

**Jenkins**

* Version: 2.375
* Purpose: Jenkins is used for Continuous Integration (CI) and Continuous Deployment (CD). It automates the process of building, testing, and deploying applications. Jenkins pulls the latest code from the GitHub repository, compiles it, runs tests to ensure functionality, and packages it into deployable formats. It supports a wide range of plugins to extend its functionality and integrate with various tools in the DevOps ecosystem.

**Nexus/Artifactory**

* Version: Latest Stable
* Purpose: Nexus and Artifactory are repositories used for storing and managing build artifacts. These artifacts include Docker images, JAR files, and other binary files generated during the build process. These repositories ensure that artifacts are versioned and can be retrieved during deployment. They provide security features such as access control and vulnerability scanning, ensuring that only approved and secure artifacts are used in production.

**Ansible**

* Version: 2.9
* Purpose: Ansible is used for configuration management and automating the provisioning and configuration of infrastructure. It uses a declarative language to define system configurations, making it easy to ensure consistency across multiple environments. Ansible can automate complex tasks such as software installation, service configuration, and user management, reducing the potential for human error and increasing operational efficiency.

**Terraform**

* Version: 1.1
* Purpose: Terraform is an Infrastructure as Code (IaC) tool used to define and provision infrastructure in a consistent and reproducible manner. It allows teams to create, update, and manage infrastructure resources like virtual machines, networks, and storage using declarative configuration files. Terraform supports multiple cloud providers and enables version control of infrastructure changes, facilitating collaboration and rollback capabilities.

**Docker**

* Version: 20.10
* Purpose: Docker is used for containerization, packaging applications and their dependencies into portable containers. This ensures that the application runs consistently across different environments, from development to production. Docker simplifies the process of setting up development environments, testing, and deployment. Containers are lightweight and isolated, providing a secure and efficient way to run applications.

**Kubernetes**

* Version: 1.21
* Purpose: Kubernetes is a container orchestration platform that manages the deployment, scaling, and operation of containerized applications. It ensures high availability and scalability by distributing containers across a cluster of nodes. Kubernetes provides features like automated rollouts and rollbacks, service discovery, load balancing, and self-healing, ensuring that the application remains robust and responsive to changes in demand.

.

**Automated Deployment Process Using Jenkins CI/CD Pipeline**

The deployment of our application is fully automated using a Continuous Integration/Continuous Deployment (CI/CD) pipeline managed by Jenkins. Here’s a detailed breakdown of how it works:

**Code Commit**

1. Developer Actions:
   * Developers write and commit code changes to the GitHub repository.
   * Each commit includes a message describing the changes and any relevant issue or feature it addresses.
2. Triggering the Pipeline:
   * A webhook configured in GitHub triggers the CI/CD pipeline in Jenkins as soon as a new commit is detected.
   * Jenkins then initiates the build process, ensuring the latest code changes are integrated and tested promptly.

**Build Stage**

1. Code Retrieval:
   * Jenkins pulls the latest code from the GitHub repository, ensuring it has the most up-to-date version of the application.
2. Build Process:
   * Jenkins uses build tools like Maven to compile the code.
   * During this stage, all dependencies are resolved, and the application is compiled into executable binaries or packages.
3. Unit Testing:
   * Jenkins runs unit tests to validate the functionality of individual components of the code.
   * Test results are analyzed, and any failures are reported back to the developers for immediate attention.

**Testing Stage**

1. Integration Testing:
   * Jenkins runs integration tests to ensure that different parts of the application work together as expected.
   * These tests verify the interactions between various components and external systems.
2. Functional Testing:
   * Functional tests are executed to validate that the application behaves correctly from an end-user perspective.
   * This stage includes automated UI tests, API tests, and other behavior-driven development (BDD) tests.
3. Packaging:
   * If all tests pass, Jenkins packages the application into deployable formats, such as JAR files, WAR files, or Docker images.

**Artifact Storage**

1. Storing Artifacts:
   * The packaged build artifacts are pushed to an artifact repository like Nexus or Artifactory.
   * These repositories manage versioning and ensure that all artifacts are stored securely and can be retrieved for deployment.
2. Version Control:
   * Each build artifact is tagged with a unique version number, facilitating easy tracking and rollback if necessary.

**Infrastructure Provisioning**

1. Provisioning with Terraform:
   * Terraform scripts are executed to provision the necessary infrastructure, including virtual machines, networks, and storage.
   * Terraform’s declarative configuration files ensure that the infrastructure is consistent and reproducible.
2. Configuration with Ansible:
   * Ansible playbooks are run to configure the provisioned infrastructure.
   * These playbooks install required software, set up configurations, and ensure that all systems are correctly initialized and ready for deployment.

**Containerization**

1. Building Docker Images:
   * Docker builds images from the application artifacts.
   * Each image includes the application and its runtime environment, ensuring consistency across different deployment environments.
2. Pushing to Docker Registry:
   * The built Docker images are tagged with version numbers and pushed to a Docker registry, such as Docker Hub or a private registry.
   * This registry acts as a central repository for all container images, making them available for deployment.

**Deployment to Kubernetes**

1. Pulling Docker Images:
   * Kubernetes pulls the required Docker images from the Docker registry based on the deployment manifests.
2. Deploying Containers:
   * Deployment manifests (YAML files) are applied to Kubernetes.
   * These manifests define how the containers should be deployed within the Kubernetes cluster, specifying details such as replicas, environment variables, and resource limits.

**Service Configuration**

1. Networking:
   * Kubernetes configures networking between services, ensuring secure and efficient communication.
   * Services are exposed via Kubernetes services and ingress controllers.
2. Reverse Proxy:
   * Nginx acts as a reverse proxy, routing external traffic to the appropriate services within the cluster.
   * This setup enhances security and performance by managing incoming requests and distributing them to the correct backend services.

**Monitoring and Logging**

1. Metrics Collection:
   * Prometheus collects metrics from the application and infrastructure, such as CPU usage, memory consumption, and request rates.
   * These metrics provide insights into the system’s performance and health.
2. Visualization:
   * Grafana visualizes the collected metrics through real-time monitoring dashboards.
   * Dashboards are customizable, allowing teams to monitor key performance indicators (KPIs) and set up alerts for critical issues.
3. Log Aggregation:
   * Logs from different components of the application and infrastructure are aggregated using centralized logging solutions.
   * These logs are analyzed to detect and diagnose issues, ensuring quick resolution and maintaining the application’s stability.

By integrating these tools and processes, our CI/CD pipeline ensures a robust, efficient, and automated deployment process that maintains high standards of code quality, reliability, and performance.