DevOps by Infosys

Course Objectives	**Course Outcomes**
• **DevOps and Agile Methodologies**: Gain a comprehensive understanding of DevOps, its importance, and how it addresses industry challenges. And also about Agile development principles, frameworks, and their integration with DevOps practices.	• Ability to implement and manage DevOps practices in an organization. Effectively use Agile methodologies to improve project delivery and team collaboration.
• **Master Python for DevOps Automation**: Develop proficiency in Python programming for automation in DevOps environments. Explore Python's data structures, functions, object-oriented programming, and essential libraries for DevOps tasks.	• Develop automated scripts and applications using Python to streamline DevOps processes. Utilize Python libraries for data manipulation, file handling, and system operations relevant to DevOps tasks.
• **Containerization and Orchestration**: Comprehend the concepts and practical skills of containerization using Docker. Learn the container orchestration with Kubernetes and OpenShift to manage and deploy scalable applications.	• Create, manage, and deploy containerized applications using Docker. Orchestrate containers using Kubernetes and OpenShift for efficient resource management and application scaling.
• **Develop Microservices and Serverless Applications**: Gain proficiency on microservices architecture, its benefits, & challenges and Implement microservices and serverless applications using modern technologies and frameworks.	• Design and develop robust microservices architectures. Utilize containerization and orchestration tools to manage microservices-based applications effectively.

• **Implement CI/CD Pipelines**: Master the principles and practices of Continuous Integration and Continuous Delivery (CI/CD). Use tools like Git, Chef, and Puppet for version control, configuration management, and automated deployments.	• Implement and maintain CI/CD pipelines to ensure seamless code integration and delivery. Utilize configuration management tools like Chef and Puppet for infrastructure as code and environment consistency.

Course Duration: 45 Hours

Course Content:

Unit 1: Introduction to DevOps, Agile Development and Scrum

Overview of DevOps - What is DevOps and why is it important? - How DevOps helps solve various Industry problems - Dev Challenges and Solution - Ops Challenges and Solution - Case Study: Deloitte New Zealand - Case Study: Daimler - Trucks NA - DevOps Market Trends - Why Agile - Agile Manifesto - Agile Principles - Agile Methodologies, Frameworks and processes - Agile Leadership - Agile with DevOps - Scrum theory and principles - Scrum Foundations (5 Scrum Values) - The Scrum Framework- The Definition of Done - Running a Scrum project - Working with people and teams - Scrum in your organization - Product Owner roles and responsibilities - Development team roles and responsibilities

Unit 2: Python for DevOps

Python Applications in DevOps - Variables, Operands and expressions - Conditional statements- Loops& Structural pattern matching - Accepting user input and eval function - Lists - Tuples - Strings manipulation - Sets and set operations - Python dictionary - User-defined functions - Function parameters and different types - Files input/output functions - Global variables - Global keyword - Lambda functions - Built-in functions - Object-oriented concepts - Public, protected and private attributes - Class variable and instance variable - Constructor and destructor - Inheritance and its types- Method resolution order - Overloading and overriding - Setter and setter methods - Standard libraries- Packages and import statements - Reload function - Creating a module - Important modules in python- Sys module - OS, Math, Date-time, Random and JSON modules - Regular expression – Exception handling - Basics of data analysis:- NumPy - Arrays: Array operations -Indexing, slicing, and Iterating - NumPy array attributes - Matrix product - NumPy functions - Array manipulation

Unit 3: Introduction to Containers w/ Docker, Kubernetes & OpenShift

Containerization - Namespaces - Docker - Docker Architecture - Container Lifecycle - Docker CLI - Port Binding - Detached and Foreground Mode - Dockerfile - Dockerfile Instructions - Docker Image - Docker Registry - Container Storage - Volumes - Docker Compose - Docker Swarm - Introduction to Container Orchestration - Kubernetes Core Concepts - Understanding Pods - ReplicaSet and Replication Controller - Deployments - DaemonSets - Rolling Updates and Rollbacks - Scaling Application - Get started with container technology - Openshift architecture - Installing openshift container platform - Openshift networking concepts - Creating applications with the openshift web console - Manage Containers - Manage Container Images - Create custom container images - Deploy containerized applications on OpenShift - Troubleshoot containerized applications

Unit 4: Application Development using Microservices and Serverless

Introduction to Microservices - Monolithic Architecture - SOA Architecture - Key benefits of Microservices - Challenges in Microservices - Introducing Microservices Architecture - Microservices Design Patterns - Use case: Apollo Store - Decomposition Strategies - Obstacles in Decomposition - Introduction of Docker with Microservices - Managing Containers in Microservices- Operations in Openshift - Streamlining Services in OpenShift

Unit 5: Continuous Integration and Continuous Delivery (CI/CD)

Version Control - Git Introduction - Git Installation - Commonly used commands in Git - Working with Remote repository - Branching and merging in Git - Merge Conflicts - Stashing, Rebasing, Reverting, and Resetting - Git Workflows - Configuration Code - Resources and Recipes - Data bags - Using community cookbooks - Chef Managed Infrastructure - ChefDK - Chef -repo - .chef Directory - Nodes - Chef Resources - Package - Resource Collection - Security Model of Chef - Introduction to Configuration Management - Introduction to Puppet - Puppet architecture - Puppet Manifest - Modules in Puppet - Server Configuration Management

INDUSTRY USE CASE:

20 Industry Use Cases

Use Case1: Microservices Deployment with Docker and Kubernetes Description:

Containerize individual microservices of an application using Docker and orchestrate their deployment, scaling, and load balancing using Kubernetes.

Tasks: All the below tasks has to be completed using Agile Methodology.

1. Identify Microservices
   - Identify the components or functionalities of the application that can be split into microservices. Determine the boundaries and responsibilities of each microservice.
2. Containerize Microservices
   - Create a Dockerfile for each microservice, specifying the base image, dependencies, and configuration required to run the microservice.
   - Build Docker images for each microservice using the Dockerfiles.
   - Push the Docker images to a container registry (e.g., Docker Hub) for easy accessibility.
3. Set Up a Kubernetes Cluster:
   - Set up a Kubernetes cluster using a platform like Minikube, kind, or a managed Kubernetes service.
   - Install and configure the Kubernetes command-line tool (kubectl) to interact with the cluster.
4. Create Kubernetes Deployment Manifests:
   - Define Kubernetes Deployment manifests for each microservice. These manifests specify the desired state of each microservice and its container image.
   - Configure resource limits, environment variables, ports, and other necessary parameters for each microservice in the manifests.
5. Create Kubernetes Service Manifests:
   - Define Kubernetes Service manifests to expose the microservices internally or externally.
   - Specify the service type (ClusterIP, NodePort, LoadBalancer), ports, and target microservice in the manifest.
6. Apply Deployment and Service Manifests:
   - Use kubectl to apply the Deployment and Service manifests to the Kubernetes cluster.
   - This will create the necessary pods (running instances of microservice

containers) and services for each microservice.

7. Validate Microservices Deployment:
   - Use kubectl commands to verify the status and health of the microservices, pods, and services.
   - Ensure that the microservices are running and accessible within the cluster.
8. Test Communication Between Microservices:
   - Set up communication between the microservices by utilizing service names or IP addresses.
   - Test the communication between microservices to ensure proper integration and functionality.
9. Scale Microservices:
   - Use Kubernetes commands or configure autoscaling policies to scale the microservices horizontally based on metrics like CPU usage or request load.
   - Monitor the scaling behavior and ensure that the microservices can handle increased traffic.

Use Case 2: Continuous Integration and Delivery Pipeline with Jenkins and Docker Description:

Set up a CI/CD pipeline using Jenkins and Docker, automating the build, testing, and deployment of applications in a Dockerized environment.

Tasks: All the below tasks has to be completed using Agile Methodology.

1. Set up Jenkins:

● Install Jenkins on a server or use a cloud-based Jenkins service.

● Configure Jenkins with necessary plugins, such as the Docker plugin, Git plugin, and Pipeline plugin.

2. Set up Source Code Repository:

● Create a version control repository (e.g., Git repository) to store the application source code.

● Connect Jenkins to the repository and configure the repository's webhook or polling mechanism to trigger Jenkins builds.

3. Configure Jenkins Job:

● Create a new Jenkins job to manage the CI/CD pipeline.

● Define the job as a Jenkins pipeline, which allows for a more flexible and scriptable approach to CI/CD.

4. Define Build Stage:

● Define a build stage in the Jenkins pipeline to pull the source code from the repository and build the Docker image.

● Configure the Docker plugin to build the Docker image based on a Dockerfile in the source code repository.

5. Configure Deployment Stage:

● Define a deployment stage in the Jenkins pipeline to deploy the Docker image to the target environment.

● This can be a staging environment, production environment, or any other deployment target.

6. Configure Deployment Tools:

● Integrate deployment tools (e.g., Kubernetes, Docker Swarm) with Jenkins to manage the deployment of Docker images.

● Use the respective plugins or command-line tools to interact with the deployment target.

7. Implement Continuous Delivery:

● Set up a continuous delivery mechanism to automatically deploy the application to the production environment after successful integration and acceptance testing.

● Ensure the pipeline includes necessary approval steps or gates to prevent unauthorized deployments.

8. Configure Notifications and Reporting:

● Configure notifications to relevant stakeholders (e.g., email notifications, messaging platforms) on build status, test results, and deployment updates.

Use Case 3: Blue-Green Deployment with Kubernetes Description:

Implement a blue-green deployment strategy using Kubernetes, allowing for seamless and zero- downtime application updates and rollbacks.

Tasks: All the below tasks has to be completed using Agile Methodology.

1. Set up Kubernetes Cluster:

● Set up a Kubernetes cluster using a platform like Minikube, kind, or a managed Kubernetes service.

● Install and configure the Kubernetes command-line tool (kubectl) to interact with the cluster. 2 Create Kubernetes Deployment Manifests:

● Define Kubernetes Deployment manifests for both the blue and green versions of the application.

● Specify the desired state of each deployment, including the container image, resource requirements, and any environment variables.

3. Create Kubernetes Service Manifests:

● Define Kubernetes Service manifests to expose the blue and green deployments internally or externally.

- Specify the service type (ClusterIP, NodePort, LoadBalancer), ports, and target deployments in the manifests.

4. Apply Blue Deployment:

- Use kubectl to apply the blue deployment manifest to the Kubernetes cluster.
- This will create the necessary pods (running instances of the blue deployment) and services for the blue version of the application.

5. Verify Blue Deployment:

- Use kubectl commands to verify the status and health of the blue deployment, pods, and services.
- Ensure that the blue version of the application is running and accessible within the cluster.

7. Create Green Deployment:

- Update the deployment manifest to reflect the green version of the application, including the updated container image or code.
- Apply the green deployment manifest to the Kubernetes cluster, creating the necessary pods and services for the green version.

8. Configure Blue-Green Service Switching:

- Utilize Kubernetes service labels, selectors, and ingress controllers to switch traffic between the blue and green deployments seamlessly.
- Set up a load balancer or ingress rule to direct traffic to the appropriate deployment.

9. Route Traffic to Green Deployment:

- Gradually shift traffic from the blue deployment to the green deployment.
- Monitor the traffic and gradually increase the percentage of traffic routed to the green deployment based on predefined criteria or testing results.

10. Validate Green Deployment:

- Monitor the health and performance of the green deployment, pods, and services.
- Conduct additional testing and quality checks to ensure the stability and functionality of the green version of the application.

11. Monitor and Rollback (if needed):

- Continuously monitor the performance, any anomalies related to the green deployment.
- If any issues or negative impact is detected, rollback to the blue deployment by redirecting traffic back to the blue deployment.

Use case 4: Deployment of a Mobile Application Description:

Create a project to build and deploy a mobile application for a fictitious company. The application aims to provide users with a platform to browse and purchase products from the company's catalog. The project will incorporate DevOps practices, Agile principles, and Scrum framework throughout its lifecycle.

Tasks: All the below tasks has to be completed using Agile Methodology.

1. Containerize the Application:

- Use Docker to containerize the mobile application and its dependencies. This ensures consistency in the development and deployment environments across different stages of the pipeline.
- Create a Dockerfile that defines the application's runtime environment, including the base image, dependencies, and any necessary configuration.

2. Implement Continuous Integration (CI):

- Set up a CI server like Jenkins or GitLab CI/CD to automate the build, test, and integration process.
- Configure the CI server to pull the application's source code from a version control repository and build a Docker image using the Dockerfile.
- Execute automated tests within the Docker container to ensure code quality and functionality.

3. Continuous Delivery and Deployment (CD):

- Use Docker images as artifacts for deployment.
- Set up a Docker registry (e.g., Docker Hub) to store and manage Docker images.
- Configure the CI server to push the Docker images to the registry once the build and tests pass successfully.
- Implement CD pipelines using tools like Jenkins, GitLab CI/CD, or Kubernetes Helm charts to automate the deployment process.
- Define deployment configurations using Docker Compose or Kubernetes manifests to specify the required services, environment variables, and networking.

4. Infrastructure Orchestration with Docker and Kubernetes:

- Utilize Docker Swarm or Kubernetes for container orchestration and management.
- Define the infrastructure requirements and configuration using declarative manifests (e.g., Docker Compose files or Kubernetes YAML files).
- Deploy the containerized application on a Docker Swarm cluster or Kubernetes cluster.
- Leverage features like scaling, load balancing, and health checks to ensure the application's availability and scalability.

5. Monitoring and Logging:

- Implement monitoring and logging tools (e.g., ELK stack) to gain insights into the application's performance, resource utilization, and logs.
- Configure the monitoring system to collect metrics and monitor the Docker containers, application performance, and infrastructure health.

Use Case 5: Deploying a Scalable Web Application with Docker, Kubernetes, and OpenShift Description:

The project involves deploying a scalable web application using Docker containers, managing orchestration with Kubernetes, and leveraging the features of OpenShift, a container platform built on Kubernetes. The web application could be a simple e-commerce website, a blogging platform, or any other application that requires scalability and high availability.

Tasks: All the below tasks has to be completed using Agile Methodology.

1. Containerizing the Web Application:

- Dockerize the web application by creating a Dockerfile that defines the application's dependencies and configurations.
- Build a Docker image for the web application, including all necessary files and dependencies.

2. Setting up Kubernetes Cluster:

- Provision a Kubernetes cluster using Minikube.
- Configure the cluster with the necessary resources and networking settings.

3. Deploying the Application on Kubernetes:

- Create Kubernetes deployment manifests that define the desired state of the application, including the Docker image, resource limits, and environment variables.
- Apply the deployment manifests to the Kubernetes cluster to create and manage the application's pods.

4. Ensuring High Availability and Scalability:

- Configure Kubernetes Horizontal Pod Autoscaler (HPA) to automatically scale the application based on metrics such as CPU utilization or incoming traffic.
- Set up Kubernetes Ingress to route incoming traffic to the application pods, allowing for load balancing and high availability.

5. Continuous Integration and Deployment:

- Integrate the project with a CI/CD tool like Jenkins or GitLab CI/CD to automate the build, test, and deployment processes.
- Create a pipeline that triggers builds upon code changes, runs tests, and deploys the updated application to the Kubernetes cluster.

6. Utilizing OpenShift Platform:

- Deploy the Kubernetes cluster on OpenShift, leveraging its additional features such as integrated container registry, image streams, and build configurations.
- Utilize OpenShift's web console or command-line tools to manage the application deployment, scaling, and monitoring.

Use Case 6: Scalable Web Application with Docker Swarm Description:

Deploy a scalable web application using Docker Swarm, allowing for easy horizontal scaling and load balancing.

Tasks: All the below tasks has to be completed using Agile Methodology.

1. Set up Docker Swarm:

- Set up a group of Docker hosts that will serve as the swarm nodes. Ensure that Docker is installed on each host.
- Choose one host to be the swarm manager.
- Note down the join token displayed after initializing the swarm.

2. Containerize the Web Application:

- Create a Dockerfile that defines the application's dependencies, configurations, and instructions to build the Docker image.
- Push the Docker image to a registry if necessary.

3: Create a Docker Compose File:

- Create a Docker Compose YAML file that describes the desired state of the application.
- Define services for each component of the web application, including the Docker image, ports, volumes, environment variables, and any necessary configurations.
- Specify the desired number of replicas for each service.

4: Deploy the Application Stack:

- Deploy the Docker stack.
- Docker Swarm will create the specified number of replicas for each service and distribute them across the swarm nodes.
- Verify that the services are running.

5: Scale the Application:

- Scale the application services and specify the desired number of replicas for each service.
- Docker Swarm will automatically adjust the number of replicas across the swarm nodes. Step 6: Load Balancing and Service Discovery:
- Docker Swarm provides built-in load balancing for services. Incoming traffic is

automatically distributed among the available replicas of the service.

- Optionally, configure an external load balancer or use Docker Swarm's built-in routing mesh to expose the application services.

7: Monitor and Manage the Swarm:

- Use Docker Swarm to monitor the state of services, replicas, and nodes.

- Utilize third-party monitoring tools or built-in Docker Swarm monitoring features to gain insights into the swarm's performance and health.

Use Case 7: Hybrid Cloud Deployment with OpenShift. Description:

Deploy an application across multiple cloud providers using OpenShift, enabling seamless deployment and management in a hybrid cloud environment.

Tasks: All the below tasks has to be completed using Agile Methodology.

1. Set Up OpenShift Cluster:

- Choose a cloud provider or an on-premises environment to host your OpenShift cluster.

- Provision the required infrastructure resources, such as virtual machines or bare-metal servers, with the necessary networking and security configurations.

- Install OpenShift on the infrastructure by following the official documentation and guidelines provided by Red Hat.

- Configure the OpenShift cluster with appropriate authentication mechanisms, user access controls, and networking settings.

2. Establish Connectivity to Public Cloud:

- Set up a connection between your on-premises OpenShift cluster and the public cloud provider of your choice.

- Configure VPN or Direct Connect services to establish a secure and reliable network connection between the on-premises environment and the public cloud.

3. Define Cluster Zones:

- Divide the OpenShift cluster into zones, separating the on-premises nodes from the cloud nodes.

- Assign labels or tags to each node based on their zone and location, enabling resource allocation and scheduling based on these attributes.

4. Configure Hybrid Networking:

- Set up networking configurations that allow communication between the on-premises environment and the public cloud.

- Configure appropriate routing, load balancing, and firewall rules to enable seamless connectivity across the hybrid cloud deployment.

5. Deploy Applications:

- Containerize your applications using Docker and create appropriate Docker images.
- Define Kubernetes deployment manifests or OpenShift deployment configurations to describe the desired state of your application, including the required resources, environment variables,and service configurations.
- Deploy the applications onto the OpenShift cluster, taking advantage of the hybrid cloud deployment capabilities.
- Use OpenShift features like Route objects or Ingress controllers to expose the applications and make them accessible from both on-premises and cloud environments.

6. Implement Hybrid Cloud Data Management:

- Define a data management strategy that encompasses both on-premises and cloud storage systems.
- Utilize storage solutions like NFS, object storage, or cloud-specific storage services to ensure seamless data access and replication across the hybrid cloud deployment.

7. Monitor and Manage the Hybrid Cloud Deployment:

- Set up monitoring and logging systems to collect metrics and logs from both on-premises and cloud environments.
- Use OpenShift monitoring and observability features or third-party tools to gain insights into the performance, health, and availability of the hybrid cloud deployment.
- Implement automation and management tools to handle tasks such as scaling, rolling updates, and backup/restoration procedures across the hybrid cloud infrastructure.

Use Case 8: Secure Containerization with Docker and Kubernetes Description:

Implement security best practices in containerization using Docker and Kubernetes, including image vulnerability scanning, pod security policies, and network policies.

Tasks: All the below tasks has to be completed using Agile Methodology.

1. Secure Docker Images:

- Use a secure base image from trusted sources like the Docker official repository or certified vendors.
- Regularly update the base image to include security patches and fixes.
- Scan Docker images for vulnerabilities.
- Avoid including unnecessary or vulnerable software packages in the Docker image.

2. Secure Docker Container Runtime:

- Utilize user namespaces and restrict container privileges to minimize the impact of any potential security breaches.

- Enable Docker Content Trust to ensure the integrity and authenticity of the pulled Docker images.
- Implement container resource limits to prevent resource abuse and ensure fair resource allocation.

3: Secure Kubernetes Cluster:

- Secure the Kubernetes cluster by following best practices like enabling RBAC (Role-Based Access Control), implementing network policies, and disabling insecure communication protocols.
- Regularly update Kubernetes components to benefit from security patches and fixes.
- Apply pod security policies to enforce security measures like limiting container capabilities, controlling host access, and setting security context constraints.

4: Secure Kubernetes Manifests:

- Implement security-focused practices while defining Kubernetes manifests.
- Avoid hardcoding sensitive information like credentials and secrets in manifests.
- Use Kubernetes secrets to manage and protect sensitive data.
- Implement resource limits and quotas to prevent resource exhaustion attacks.

5: Secure Container Communication:

- Utilize secure communication channels like HTTPS or TLS for inter-container communication and API interactions.
- Implement network isolation and segmentation to prevent unauthorized access to containers.
- Leverage Kubernetes network policies to control ingress and egress traffic between containers and pods.

6: Secure Secrets Management:

- Store sensitive information like credentials, API keys, and certificates securely using Kubernetes Secrets, HashiCorp Vault, or other secure storage solutions within the cluster.

Use Case 9: Provisioning and Configuring a Web Server with Chef Description:

Use Chef, an infrastructure automation tool, to provision and configure a web server. Set up a basic web server using Apache HTTP Server and configure it to serve a static website.

Tasks: All the below tasks has to be completed using Agile Methodology.

1. Set up the Chef Environment:

- Install and configure the Chef Workstation on your local machine.
- Set up a Chef Server or use the Chef Automate hosted service for managing

infrastructure and storing configuration data.

2. Create a Cookbook:

- Create a new Chef cookbook chef command.
- Define the cookbook's structure, including recipes, attributes, templates, and files.

3. Write Recipes:

- Create a recipe within the cookbook to install and configure the Apache HTTP Server.
- Specify the necessary resources, such as packages, services, and configuration files, in the recipe.
- Define attributes to customize the web server configuration, such as port number, virtual hosts, or SSL settings.

4. Test the Cookbook Locally:

- Use Test Kitchen, a testing framework for Chef, to test the cookbook locally.

5. Upload the Cookbook to the Chef Server:

- Upload the cookbook to the Chef Server.
- Ensure the cookbook is available on the Chef Server for deployment to target nodes.

6. Set up the Target Node:

- Prepare the target node by installing the Chef client and configuring it to connect to the Chef Server.
- Configure the node's run-list to include the recipe for provisioning the web server.

7. Deploy the Cookbook:

- Use the Chef client to apply the cookbook to the target node.
- The Chef client will connect to the Chef Server, download the cookbook, and apply the defined resources on the node.

8. Verify the Web Server Configuration:

- Access the target node and verify that the Apache HTTP Server is installed and running.
- Check the web server's configuration files and ensure they match the desired settings defined in the cookbook.

Use Case 10: Configure Nginx Web Server with Puppet

Description: Use Puppet, an infrastructure automation tool, to configure a Nginx web server. Define

Puppet manifests to install Nginx, manage its configuration files, and start the web server.

Tasks: All the below tasks has to be completed using Agile Methodology.

1. Set up the Puppet Environment:

- Install and configure the Puppet Agent on the target server where Nginx will be configured.
- Set up the Puppet Master server or use a hosted Puppet environment for managing the Puppet infrastructure.

2. Create Puppet Manifests:

- Create a Puppet module or class for managing the Nginx web server configuration.
- Define the necessary resources, such as packages, services, configuration files, and directories, within the Puppet manifests.

3. Install Nginx Package:

- Use the resource in Puppet to install the Nginx package on the target server.
- Specify the package name and version, ensuring that it matches the desired version.

4. Manage Nginx Configuration Files:

   - Create Resources in Puppet to manage Nginx configuration files.
   - Specify the desired content of the configuration files using Puppet's template or file content functionality.

5. Assign the Manifest to the Target Node:

   - Define a Puppet node definition or classification to assign the Nginx configuration manifest to the target node.
   - Specify the node's hostname or other identifying factors to apply the configuration to the correct target server.

6. Verify the Nginx Configuration:

   - Access the target server and verify that Nginx is installed and running.
   - Check the Nginx configuration files to ensure they match the desired settings defined in the Puppet manifest.

7. Manage Configuration Updates:

   - Make changes to the Puppet manifest as needed to update the Nginx configuration.
   - Deploy the updated manifest to the target server using the Puppet Agent to apply the changes.

Use Case 11: Implement Firewall Rules with Puppet Description:

Employ Puppet to implement firewall rules on a target server. Define Puppet manifests to install and configure a firewall management tool (e.g., iptables), define firewall rules, and ensure the firewall is enabled.

Tasks: All the below tasks has to be completed using Agile Methodology.

1. Set up the Puppet Environment:

- Install Puppet on the Puppet Master server and Puppet Agent on the target server where the firewall rules will be implemented.
- Configure the Puppet Master server to manage the Puppet Agent and define the Puppet environment.

2. Create the Puppet Manifests:

- Create a new Puppet module for managing firewall rules. This module will contain the necessary manifests and files for configuring the firewall.
- Inside the module, define a manifest file to describe the desired state of the firewall rules.

3. Install and Configure Firewall Management Tool:

- Define a package resource in the manifest to install the firewall management tool using the appropriate package manager for your operating system.
- Ensure the package is installed and any required dependencies are resolved.
- Configure the firewall management tool to start at system boot time.

4. Define Firewall Rules:

- Define firewall rules in the manifest using the syntax supported by the chosen firewall management tool.
- Specify the desired firewall rules, including allowed ports, protocols, IP addresses, or ranges.
- Ensure the rules are written in a way that follows best practices for security and compliance.

5. Enable the Firewall:

- Define a service resource in the manifest to manage the firewall service.
- Use Puppet's service management capabilities to enable and start the firewall service.

Use Case 12: Web Scraping and Data Analysis Automation Description:

Create a DevOps pipeline to automate the web scraping and data analysis process using Python libraries such as Pandas, NumPy, and BeautifulSoup. The pipeline will consist of several stages, including data extraction, transformation, and analysis. Use Docker to containerize the application and ensure its portability and reproducibility.

Tasks: All the below tasks has to be completed using Agile Methodology.

1. Set up the Development Environment:

- Install Python and the necessary libraries: Pandas, NumPy, and BeautifulSoup.
- Set up a code editor or an integrated development environment (IDE) for Python development.
- Create a project directory and initialize a Git repository for version control.
- Web Scraping:
- Use the BeautifulSoup library to extract data from a target website.
- Write Python code to scrape the desired information from the web page.
- Save the scraped data in a structured format, such as CSV or JSON.

3. Data Transformation and Cleaning:

- Load the scraped data into a Pandas DataFrame for further processing.
- Use Pandas functions to clean and transform the data, such as removing duplicates, handling missing values, or converting data types.
- Perform any necessary data preprocessing tasks, such as feature engineering or data normalization.

4. Data Analysis:

- Utilize NumPy and Pandas functions to perform data analysis and calculations.
- Generate statistical summaries, calculate aggregations, or create visualizations to gain insights from the data.
- Implement data analysis algorithms or machine learning models if relevant to the project.

5. Containerize the Application with Docker:

- Write a Dockerfile to define the Docker image for your application.
- Specify the necessary dependencies, including Python and the required libraries.
- Set up the container environment and define the entry point for running the application.

6. Build and deploy the Docker Image:

- Build the Docker image using the Dockerfile.
- Push the Docker image to a container registry for version control and sharing.
- Deploy the Docker image to a target environment, such as a local Docker engine or a cloud- based container platform.

7. Automation and CI/CD:

- Set up a CI/CD pipeline using a continuous integration and continuous deployment tool, such as Jenkins or GitLab CI/CD.

- Configure the pipeline to automatically trigger builds, tests, and deployments when changes are pushed to the repository.
- Ensure the pipeline includes automated tests for the web scraping, data transformation, and analysis processes.

Use Case 13: Machine Learning Model Deployment with Flask and Docker Description:

Develop a Flask web application to serve a trained machine learning model. Containerize the application with Docker for easy deployment. Implement RESTful APIs to accept input data, apply the model for predictions, and return the results.

Tasks: All the below tasks has to be completed using Agile Methodology.

1. Preparing the Machine Learning Model

● Train and finalize your machine learning model using Python libraries such as scikit-learn.

2. Building the Flask Application

● Create a new directory for your Flask application.

● Set up a virtual environment and install Flask and any other necessary dependencies.

● Create a Flask application script that will serve as the entry point for the application.

Define the necessary routes and endpoints for handling incoming requests and executing predictions using the trained model.

● Load the serialized machine learning model into memory when the application starts.

3. Testing the Flask Application

● Run the Flask application locally to ensure that it is working correctly.

● Send test requests to the defined endpoints and verify that the model predictions are returned as expected.

4. Creating a Dockerfile

● Create a Dockerfile in the project directory.

● Define the base image that will be used for the Docker container, typically a lightweight Python image.

● Copy the necessary files, including the Flask application script and any required model files, into the Docker container.

● Install the required dependencies using pip or any other package manager.

● Specify the command that should be executed when the Docker container starts, which is typically running the Flask application.

5. Building the Docker Image

- Build the Docker image using the Dockerfile.

Verify that the Docker image was successfully built by listing the available Docker images.

6. Running the Docker Container

- Run the Docker container locally.
- Specify any necessary port mappings if the Flask application is listening on a specific port.
- Test the deployed machine learning model by sending requests to the appropriate endpoint.

7. Deploying the Docker Image

- Push the Docker image to a container registry, such as Docker Hub or a private registry, for availability and sharing.
- Pull the Docker image from the container registry into the deployment environment.
- Configure the necessary network settings, environment variables, and other deployment configurations.
- Deploy the Docker container in the target environment and ensure it is running correctly.

Use Case 14: Microservices Application Deployment with OpenShift Description:

Create a simple microservices-based application and deploy it using OpenShift, a container orchestration platform. The application will consist of multiple microservices that communicate with each other to perform different functionalities. Each microservice will be deployed as a separate container within the OpenShift cluster, allowing for scalability, resilience, and easy management.

Tasks: All the below tasks has to be completed using Agile Methodology.

1. Designing the Microservices Architecture

Identify the different functionalities of your application and determine how to divide them into individual microservices.

- Define the APIs and communication patterns between the microservices.

2. Setting up an OpenShift Cluster

- Install and configure an OpenShift cluster either locally using Minishift or on a cloud provider like AWS or Azure.
- Set up authentication and access controls to manage cluster security.

3. Containerizing Microservices

- Containerize each microservice using Docker. Create Dockerfiles for each microservice that define the necessary dependencies and configurations.

- Build Docker images for each microservice and push them to a container registry accessible by your OpenShift cluster.

4. Defining OpenShift Deployment Configuration

- Define the necessary Kubernetes deployment manifests for each microservice. These manifests describe the desired state of the microservices, including container images, resource requirements, and network configurations.

- Create a deployment configuration for each microservice, specifying the desired number of replicas and any scaling policies.

5. Deploying Microservices on OpenShift

- Use the OpenShift command-line interface (CLI) or web console to create projects or namespaces to isolate your application.

- Deploy each microservice using the deployment configuration manifests. OpenShift will automatically create and manage the required containers within the cluster.

- Verify that the microservices are running correctly and can communicate with each other within the OpenShift cluster.

Use Case 15: Version Control and Continuous Integration with Git and Docker Description:

Create a version control system using Git and leverage its integration with Docker for continuous integration. Create a simple web application, manage its source code using Git, and automate the build and deployment process using Docker.

Tasks: All the below tasks has to be completed using Agile Methodology.

1. Setting up the Git Repository

- Create a new Git repository on a Git hosting platform such as GitHub.

- Clone the repository to your local development environment using the Git command-line interface.

2. Developing the Web Application

- Create a simple web application using your preferred programming language (e.g., Python, Node.js, Ruby).

- Initialize the project directory as a Git repository.

- Add the necessary source code files to the repository.

- Commit the initial codebase and provide a descriptive commit message.

3. Configuring Continuous Integration with Git Hooks

- Set up a Git hook, such as a post-receive hook, to trigger the CI process whenever new code is pushed to the repository.

- Write a script that executes the necessary commands to build and test the Docker image.

- Configure the Git hook to execute the script when changes are detected.

4. Setting up a CI Server

- Set up a CI server, such as Jenkins or GitLab CI/CD, that will handle the CI workflow.
- Configure the CI server to monitor the Git repository for changes and trigger the build process.
- Define a build pipeline or job that executes the necessary commands to build and test the Docker image.

5. Building and Testing the Docker Image

- Configure the CI server to execute the Docker build command, using the Dockerfile defined in the repository.
- Execute any necessary tests or validation checks on the Docker image to ensure its functionality.
- Generate relevant build artifacts, such as logs or reports, for further analysis.

6. Publishing the Docker Image

- Set up a container registry, such as Docker Hub or a private registry, to store the Docker images.
- Configure the CI server to push the built Docker image to the container registry.
- Tag the Docker image with an appropriate version or tag for easy identification and tracking.

7. Deploying the Docker Image

- Configure the deployment environment, such as a testing or production server, to pull` the Docker image from the container registry.
- Set up a deployment pipeline or process to deploy the Docker image to the desired environment.
- Automate the deployment process to ensure a smooth and consistent deployment experience.

Use Case 16: Managing Application Replicas with Kubernetes Replicase. Description:

Create and manage application replicas using Kubernetes ReplicaSet. ReplicaSet is a Kubernetes resource that ensures a specified number of replicas (pods) of a pod template are running at all times. Scale and manage application replicas effectively in a Kubernetes cluster.

Tasks: All the below tasks has to be completed using Agile Methodology.

1. Set Up the Kubernetes Cluster

- Install and set up a Kubernetes cluster using a platform like Minikube or a cloud provider's Kubernetes service.
- Verify the cluster's health and connectivity to ensure it is ready for deploying applications.
- Configure the command-line interface (CLI) tools, such as kubectl, to interact with the Kubernetes cluster.

2. Create a Pod Template

- Define a pod template specification in a YAML file that describes the desired configuration of the application pod.
- Specify the container image, ports, environment variables, and other necessary settings in the pod template.
- Ensure the pod template is ready for scaling by including appropriate resource limits and requests.

3. Create a ReplicaSet

- Write a ReplicaSet manifest in a YAML file, specifying the desired number of replicas and the pod template to use.
- Apply the ReplicaSet manifest to the Kubernetes cluster using the kubectl command.
- Verify that the ReplicaSet is created and the desired number of pods are running by checking the status of the pods and the ReplicaSet itself.

4. Scale the Application Replicas

- Update the ReplicaSet manifest to increase or decrease the desired number of replicas.
- Apply the updated ReplicaSet manifest to the cluster using the kubectl command.
- Observe the Kubernetes scheduler creating or terminating pods to maintain the desired number of replicas.

5. Perform Rolling Updates

- Modify the pod template specification to introduce changes or updates to the application, such as a new container image or environment variable.
- Apply the updated pod template to the existing ReplicaSet by modifying the ReplicaSet manifest or using the kubectl command.
- Observe the ReplicaSet orchestrating a rolling update, gradually replacing the old pods with the updated pods without causing downtime.

Use Case 17: Implementing Security Model in Chef Description:

Project focuses on the security aspects of using Chef for configuration management. Implement a security model in Chef to ensure that the infrastructure and configurations are protected from unauthorized access and adhere to security best practices.

Tasks: All the below tasks has to be completed using Agile Methodology.

1. Infrastructure Hardening

● Identify security best practices and guidelines relevant to your infrastructure.

● Implement security configurations, such as firewall rules, access controls, and secure network configurations, using Chef resources.

● Apply the necessary security patches and updates to the infrastructure components using Chef recipes or cookbooks.

2. Secure Credential Management

● Implement secure credential management using Chef data bags or encrypted data bags.

● Store sensitive information, such as passwords or API keys, securely within the Chef environment.

● Ensure that only authorized users or nodes can access and decrypt the sensitive data.

3. Role-Based Access Control (RBAC)

● Define roles and permissions for different user types within the Chef server.

● Assign appropriate permissions to users or teams based on their responsibilities and access requirements.

● Implement RBAC policies using Chef's built-in authentication and authorization mechanisms.

4. Secure Cookbook Development

● Follow secure coding practices while developing Chef cookbooks, including input validation, secure file permissions, and avoiding hardcoded secrets.

● Use trusted sources for cookbook dependencies and ensure that they are regularly updated to address security vulnerabilities.

● Perform security testing and code reviews of the cookbooks to identify and mitigate potential security issues.

5. Continuous Compliance Monitoring

● Implement Chef InSpec to define and enforce compliance policies for your infrastructure.

● Write InSpec profiles to validate the security configurations and settings of the managed nodes.

● Run regular compliance checks and monitor the results to ensure continuous compliance with security standards.

6. Secure Communication and Data Encryption

● Configure secure communication channels between the Chef server and nodes using TLS/SSL certificates.

● Enable encryption for Chef data traffic to protect sensitive information during transmission.

● Regularly review and update the SSL certificates used for secure communication.

7. Auditing and Logging

● Enable auditing and logging features in Chef to capture relevant security events and activities.

● Configure log aggregation and analysis tools to monitor and detect any security-related incidents.

● Regularly review and analyze the logs to identify potential security threats or vulnerabilities.

Use Case 18: Automated Deployment Script with Python Description:

Create a Python automation script to streamline the deployment process of a web application. The script will handle tasks such as pulling code from a version control repository, building and packaging the application, and deploying it to a target environment. This project will help you understand how Python can be used to automate and simplify the deployment process in a DevOps context.

Tasks: All the below tasks has to be completed using Agile Methodology.

1. Set Up the Project Environment

● Create a new directory for your project and set up a virtual environment using a tool like virtualenv or conda.

● Install the necessary Python packages and dependencies for the deployment script, such as GitPython for interacting with the version control repository and any specific libraries required for your deployment process.

2. Clone the Source Code Repository

● Use the GitPython library to clone the source code repository of your web application.

● Provide the necessary authentication details, such as username and password or SSH key, if required.

3. Build and Package the Application

● Define the build process for your web application, including compiling assets, running tests, and generating the necessary artifacts.

● Use Python subprocess or similar libraries to execute the build commands and capture the output.

4. Deploy the Application

- Define the deployment process for your web application, such as copying files to the target server, configuring environment variables, and restarting services.

- Utilize SSH libraries or tools like Fabric to connect to the target server and execute the deployment commands.

Use Case 19: Dockerize Machine Learning Model Deployment with Python Description:

Create a Dockerized machine learning model deployment solution using Python, Docker, and relevant machine learning libraries.

Tasks: All the below tasks has to be completed using Agile Methodology.

1. Set Up the Project Environment

- Create a new directory for your project and set up a virtual environment using a tool like virtualenv or conda.

- Install the necessary Python packages for developing and deploying your machine learning model, such as scikit-learn, pandas, and Flask.

2. Develop and Train the Machine Learning Model

- Write Python code to develop and train your machine learning model using the desired libraries and datasets.

- Evaluate and validate the model's performance using appropriate metrics and techniques.

3. Containerize the Machine Learning Model with Docker

- Create a Dockerfile that specifies the necessary dependencies and configuration for running your machine learning model.

- Build a Docker image based on the Dockerfile, which includes the required Python libraries, the trained model, and any other dependencies.

4. Develop the Web Service for Model Deployment

- Create a Flask application that exposes an API endpoint for making predictions using the trained machine learning model.

- Define the routes and request handling functions to accept input data, preprocess it if necessary, and return the predictions.

5. Deploy the Dockerized Model using Docker Compose

- Write a Docker Compose configuration file that describes the services needed for deployment, including the machine learning model container and any additional services required, such as a database or message broker.

- Use Docker Compose to deploy and orchestrate the containers, ensuring they communicate properly.

6. Scaling and Load Balancing

- Explore options for scaling the deployed model to handle increased traffic or workload. This could involve replicating containers, using orchestration tools like Kubernetes, or implementing load balancing strategies.

● Test the scalability and load balancing capabilities of the deployment solution to ensure it can handle increased demand.

7. Monitoring and Logging

● Set up monitoring and logging mechanisms to track the performance, usage, and errors of the deployed model.

● Implement appropriate logging practices and integrate with monitoring tools to gather insights into the model's behavior and identify potential issues.

Use Case 20: Deployment of an Application with Chef and Kubernetes Description:

Deploy an application using Chef and Kubernetes. Use Chef to automate the provisioning and configuration of the application's infrastructure, and Kubernetes to manage the deployment and scaling of the application.

Tasks: All the below tasks has to be completed using Agile Methodology.

1. Set Up the Project Environment

● Install and configure Chef on your local machine.

● Set up a Kubernetes cluster locally using tools like Minikube or on a cloud provider.

2. Define Infrastructure as Code with Chef

● Create Chef recipes and cookbooks to define the desired state of the infrastructure required for the application.

● Write recipes to install and configure dependencies, such as databases, web servers, or load balancers.

● Use Chef's resource providers to manage infrastructure resources, such as creating Kubernetes deployments or services.

3. Create Kubernetes Manifests

● Define Kubernetes deployment manifests for the application, specifying the container image, environment variables, ports, and resource requirements.

● Write Kubernetes service manifests to expose the application to external traffic and enable communication between different application components.

● Utilize Kubernetes secrets for managing sensitive information, such as API keys or database credentials.

4. Automate Deployment with Chef

● Use Chef's orchestration capabilities to automate the deployment of the application to the Kubernetes cluster.

- Write Chef recipes to apply the Kubernetes manifests and configure the desired state of the cluster.
- Utilize Chef's idempotent nature to ensure that the application remains in the desired state even after updates or scaling events.

5. Test and Validate the Deployment

- Verify that the application is successfully deployed and accessible within the Kubernetes cluster.

Test the functionality of the deployed application and validate its behavior under different scenarios, such as high traffic or scaling events.

6. Scaling and Maintenance

- Explore options for scaling the application based on traffic demands or resource utilization metrics.
- Utilize Chef and Kubernetes to scale the application horizontally by increasing the number of replicas or vertically by adjusting resource limits.
- Perform maintenance tasks, such as rolling updates or configuration changes, using Chef to ensure minimal downtime.