1. The Ansible playbooks and roles you would create for each component

Here's a summary of the Ansible playbooks and roles created for each component of your 3-tier web application deployment:

Playbooks:

**-------------**

deploy\_app.yml: Deploys the entire 3-tier web application by including the following roles:

-- load\_balancer

-- web\_frontend

-- app\_servers

-- database

create\_load\_balancer.yml: Creates the Kubernetes Service for the load balancer using the load\_balancer role.

create\_web\_frontend.yml: Creates the Kubernetes Deployment for web frontends using the web\_frontend role.

create\_app\_servers.yml: Creates the Kubernetes Deployment for application servers using the app\_servers role.

create\_database.yml: Creates the Kubernetes Deployment for the database using the database role.

update\_app.yml: Performs a rolling update of the application servers using the app\_servers role to update the Docker image version.

Roles:

**-----------**

-- common (Shared Configuration):

Contains common tasks and configurations shared across all components.

-- load\_balancer (Load Balancer Configuration):

Defines tasks for creating a Kubernetes Service for the load balancer.

Utilizes templates for load balancer service YAML.

-- web\_frontend (Web Frontend Configuration):

Defines tasks for creating a Kubernetes Deployment for web frontends.

Utilizes templates for frontend deployment YAML.

-- app\_servers (Application Servers Configuration):

Defines tasks for creating a Kubernetes Deployment for application servers.

Utilizes templates for app servers deployment YAML.

-- database (Database Configuration):

Defines tasks for creating a Kubernetes Deployment for the database.

Utilizes templates for database deployment YAML.

2. How would you organize the code for reusability across environments

To organize the Ansible code for reusability across different environments (e.g., dev, prod, stage), we've set up a directory structure and configuration approach that allows us to manage environment-specific variables and inventory files. Here's how we've organized the code:

Directory Structure:

**--------------------------**

architrave\_k8s\_app/

├── inventories/

│   ├── dev/

│   │   └── hosts

│   ├── stage/

│   │   └── hosts

│   └── prod/

│       └── hosts

├── roles/

│   ├── common/

│   ├── load\_balancer/

│   ├── web\_frontend/

│   ├── app\_servers/

│   └── database/

├── playbooks/

└── ansible.cfg

We have separate inventory directories for each environment (dev, stage, prod), allowing us to define different sets of target hosts or clusters for different stages of deployment.

Inventory Files:

**----------------------**

Within each environment's inventory directory, we've created hosts files that specify the target hosts or clusters for that environment. This separation enables you to target different k8s clusters or nodes for different stages (dev/stage/prod).

Role-Based Structure:

**-----------------------**

Roles are organized within the roles/ directory. Each role encapsulates the configuration and tasks related to a specific component of your application (e.g., load\_balancer, web\_frontend, app\_servers, database).

Environment-Specific Variable Configuration:

**--------------------------------------------------**

Inside our playbooks, we have set environment-specific variables. For example, in the create\_app\_servers.yml playbook, we defined variables like app\_servers\_replicas, app\_server\_image, etc., which can be customized for each environment.

Ansible Configuration File (ansible.cfg):

**-------------------------------------------**

We've included an ansible.cfg file in the root of our project. This file specifies global configurations for Ansible, such as the roles path, SSH settings, and other options.

By adopting this structure, we can easily manage and deploy our application to different environments by selecting the appropriate inventory and by configuring the playbook variables according to the specific requirements of each environment. This approach promotes code reusability and ensures that we can apply the same Ansible playbooks and roles with minor adjustments across different stages or clusters while keeping our configurations organized and separated.

3. Your approach to configuring other Kubernetes objects like services

To configure Kubernetes objects like Services, we've used Ansible templates within each role to define the Kubernetes resource specifications.

Service Configuration Template (roles/load\_balancer/templates/load-balancer-service.yml.j2):

**--------------------------------------------------------------------------------------------**

-- In this template, we've created a Kubernetes Service named "load-balancer" that targets pods with the label app: load-balancer.

-- The load\_balancer\_target\_port variable is used to specify the target port to which the load balancer should forward traffic.

Playbook (playbooks/create\_load\_balancer.yml):

**--------------------------------------------------**

In this playbook, we included the load\_balancer role and set the load\_balancer\_target\_port variable to define the target port.

Similarly, You have a similar structure for configuring the Kubernetes Service for the web frontends [Web Frontend Role (roles/web\_frontend)], application servers [App Servers Role (roles/app\_servers)] & database[Database Role (roles/database)].

So, by using Ansible templates and playbook variables, we can easily customize and configure Kubernetes objects like Services for each component of our application. This approach allows us to parameterize our configurations and reuse them across different environments and stages of deployment.

4. How you would manage app configuration

Managing application configuration typically involves handling environment-specific configuration settings, secrets, and other variables that our application needs to function correctly. Here's how we can manage app configuration in the context of your Ansible setup:

Configuration Files:

**-----------------------------**

Create separate configuration files for our application, each tailored to a specific environment (e.g. dev-config.yaml, stage-config.yaml, prod-config.yaml).

Store these configuration files in a separate repository or a central location accessible to our Ansible playbook.

Ansible Variables:

**---------------------**

Define Ansible variables in our playbooks for each environment. For example, we can define a variable like "app\_config\_file" that specifies the path to the environment-specific configuration file.

---

- name: Deploy 3-Tier Web Application

  hosts: k8s-cluster

  become: yes

  roles:

    - load\_balancer

    - web\_frontend

    - app\_servers

    - database

  vars:

    load\_balancer\_target\_port: 80

    frontend\_replicas: 3

    frontend\_image: my-web-frontend-image:latest

    app\_servers\_replicas: 3

    app\_server\_image: my-app-server-image:latest

    database\_username: mydbuser

    database\_port: 5432

    database\_replicas: 1

    database\_image: my-database-image:latest

    app\_config\_file: dev-config.yaml  # Environment-specific configuration file

Application Deployment:

**------------------------------**

During the deployment process (e.g. in our create\_app\_servers.yml playbook), you can use Ansible tasks to copy the environment-specific configuration file to the appropriate location within our application containers.

- name: Copy App Config to Application Servers

  copy:

    src: "{{ app\_config\_file }}"

    dest: /app/config.yaml

  become: yes

Here, Ansible copies the dev-config.yaml file to /app/config.yaml within our application servers. We need to adjust the destination path as needed to match our application's expectations.

Application Consumption:

**-------------------------------**

Ensuring that our application code is designed to read its configuration from the specified path (e.g. /app/config.yaml).

By following this approach, we can manage our application configuration separately for each environment, and Ansible helps ensure that the correct configuration is deployed along with our application code. This way, we can maintain consistency and easily switch between configurations when deploying to different environments (e.g., dev, stage, prod) without modifying the application code itself.

*\*\**\*\* I didn't make this exact changes in my code as I have already mentioned the variables & it would take little more time & I need to submit this assignment by today!!

5. Your strategy for rolling updates and minimizing downtime:

Kubernetes Deployment Updates:

**----------------------------------**

We have already included RollingUpdate strategy, where Kubernetes will ensure that only one new replica is created at a time (maxSurge: 1) and that no more than one old replica is unavailable at any given time (maxUnavailable: 0), minimizing downtime during updates.

Health Checks and Readiness Probes:

**-------------------------------------**

By implementng readiness and liveness probes in our container definitions to ensure that Kubernetes waits until the new pods are ready before scaling down the old ones. We have already included Liveness & Readiness Probes. Kubernetes will perform readiness and liveness checks by sending HTTP requests to the /healthz endpoint. We need to adjust the path and port as needed to match our application.

Rolling Update Execution:

**-----------------------------**

By running the update\_app.yml playbook, which triggers a rolling update for our application servers.

We need to ensure that the Deployment and Pod templates in our Kubernetes Deployment YAML files reference the updated image version.

Monitoring and Verification:

**--------------------------------**

Monitoring the rolling update process using ArgoCD, FluxCD, Grafana, EFK/ELK, Kubernetes dashboard, kubectl commands, or other monitoring tools to ensure that new pods are created and old pods are terminated correctly.

Verifying that our application remains responsive and stable during the update.

Rollback Strategy:

**-------------------------**

Preparing a rollback strategy in case the update encounters issues. This may involve reverting to the previous image version or configuration settings.

*\*\**\*\* I have just mentioned k8s application rolling updates and minimizing downtime using normal best practices but all these steps can become much easier if we use Kustomize or Helm or Both of Kustomize & Helm together. The implementation & execution steps would take some time to write. I can explain about this whole steps if you want me to explain about it.