FastAPI Application Deployment on AWS & Kubernetes

Overview

This documentation details the end-to-end process of deploying a FastAPI application designed to count and return the number of HTTP requests. By integrating AWS services and Kubernetes, the deployment demonstrates scalability, cost-efficiency, and adherence to DevOps best practices.

---

1. Application Overview

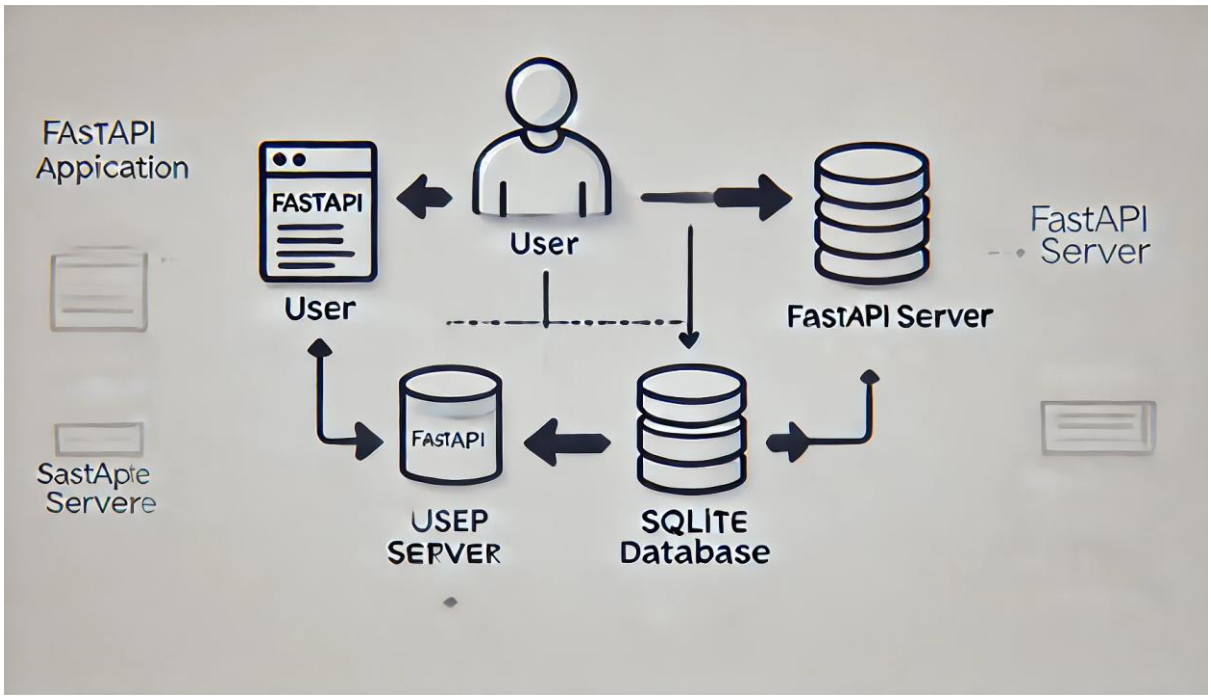
Objective

Develop a lightweight, containerized FastAPI application with a `/count` endpoint that:

- **POST /count**: Increments the request count.
- **GET /count**: Returns the current count in JSON format.

Key Considerations

- Use a database (e.g., SQLite) for persisting the count.
- Ensure modular design for scalability in production environments.



The architecture diagram illustrates the flow of requests in a FastAPI application. A user (client) sends a request to the `/count` endpoint, which is handled by the FastAPI server. The server interacts with the SQLite database to retrieve or update the count. Once the database responds, the server formats the result and returns it to the user, completing the cycle. The diagram highlights the seamless communication between the user, server, and database.

---

2. Containerization with Docker

Why Docker?

Docker standardises the application's runtime environment, ensuring consistent performance across development and production systems.

Key Considerations

- Use a minimal base image to reduce vulnerabilities and improve performance.
- Manage dependencies through a `requirements.txt` file.

Challenges

- Managing database persistence within a stateless container.
- Debugging compatibility issues with specific Python libraries.

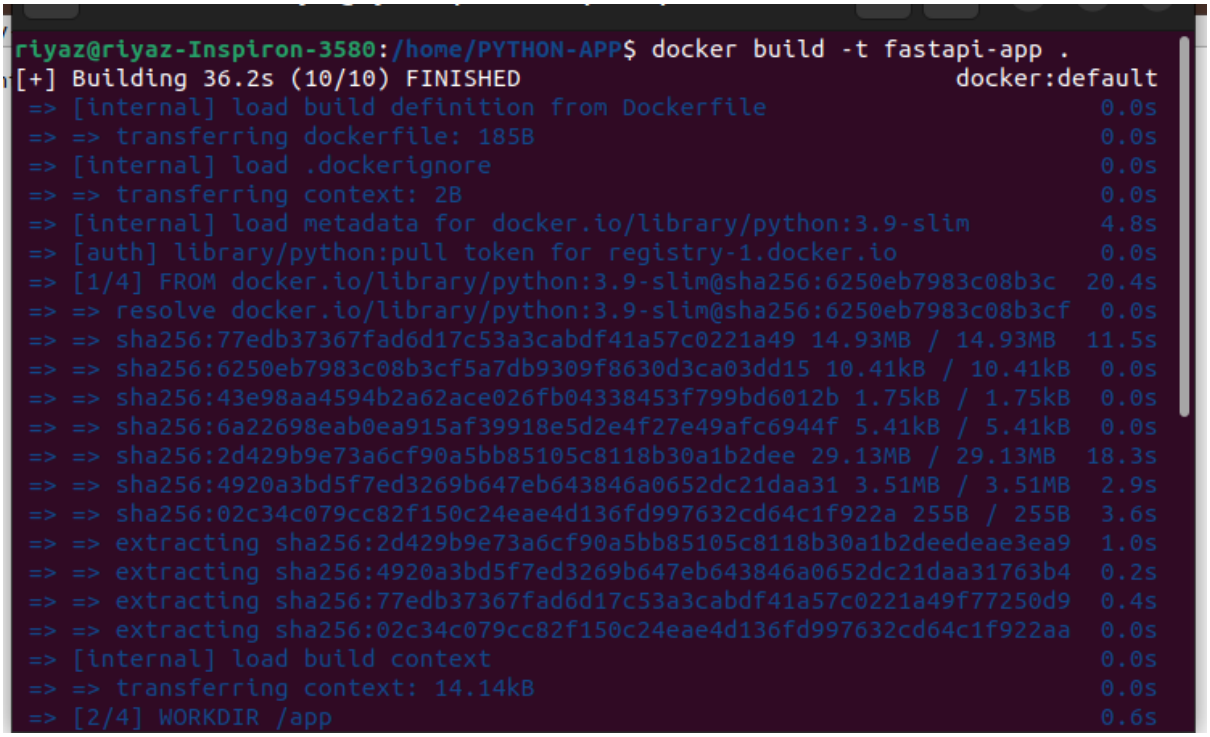
Solutions

- Configured volume mounts for persistent storage.
- Verified library compatibility using `pip check` before containerization.

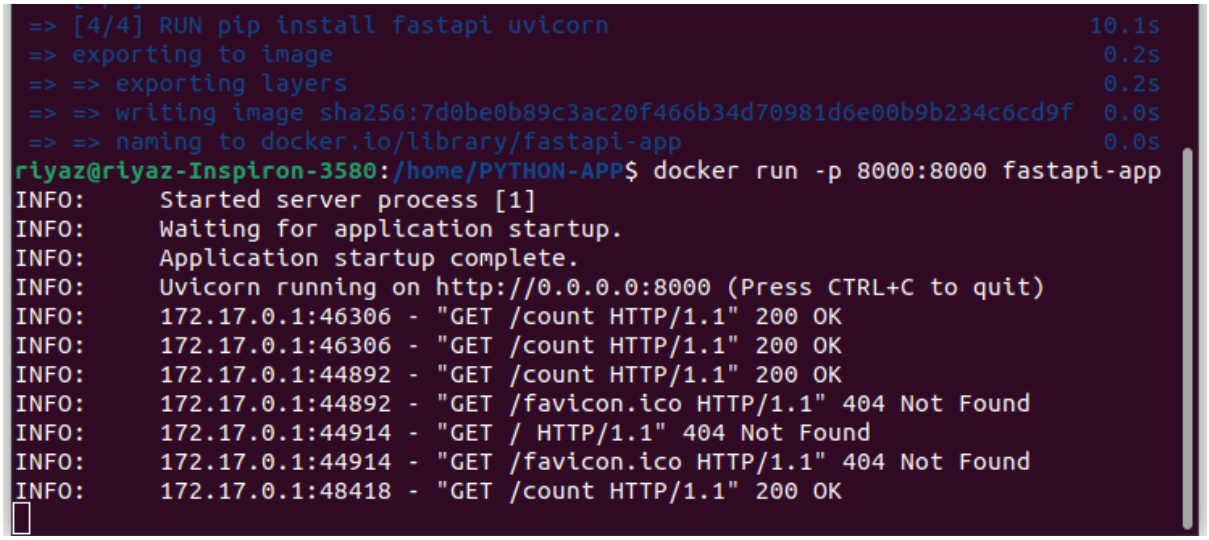
Dockerfile Screenshot

```
1  FROM python:3.9-slim
2
3  WORKDIR /app
4
5  COPY . .
6
7  RUN pip install fastapi uvicorn
8
9  CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]
```

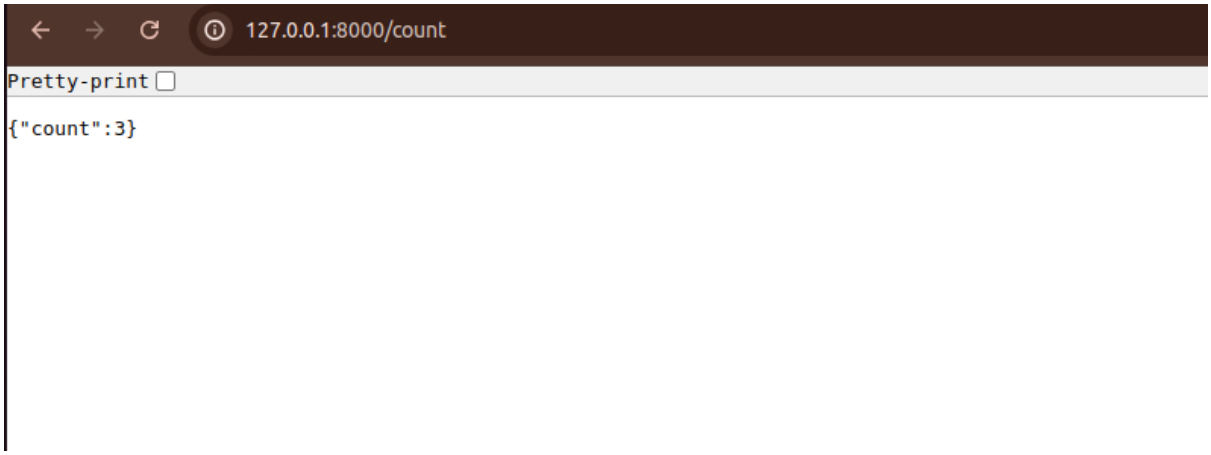
Building the Docker image:

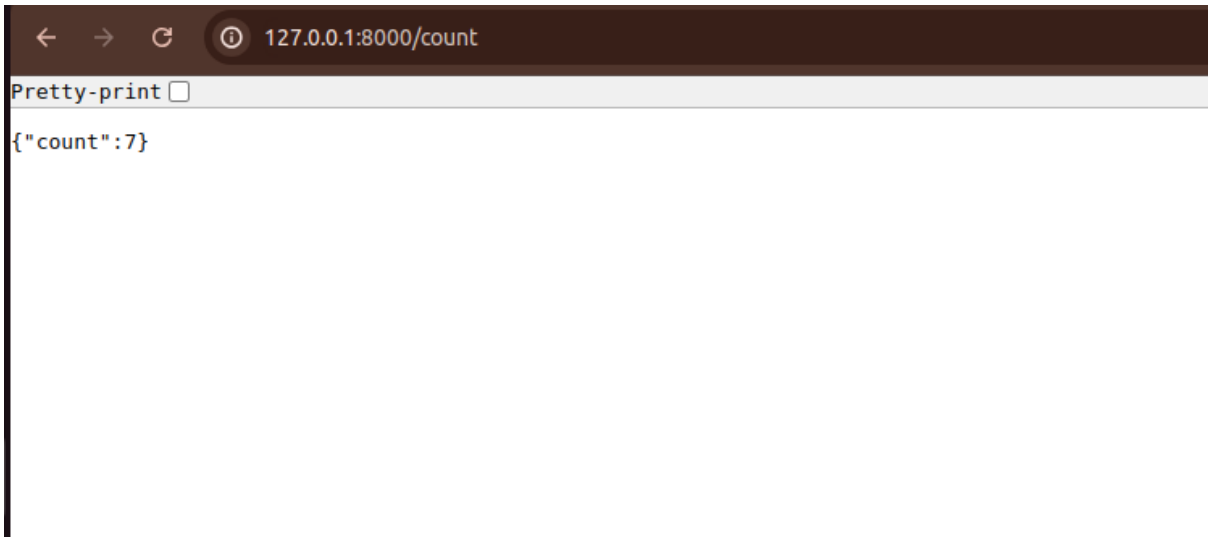


Running the Docker image:



Accessing the application:





---

3. Local Testing with Docker Compose

Objective

Simulate a multi-container environment to test the application's functionality before deploying to the cloud.

Key Considerations

- Use a `docker-compose.yaml` file to define the application and database containers.
- Map ports to allow easy access to the application.

Challenges

- Networking issues between containers.
- Data loss during container restarts.

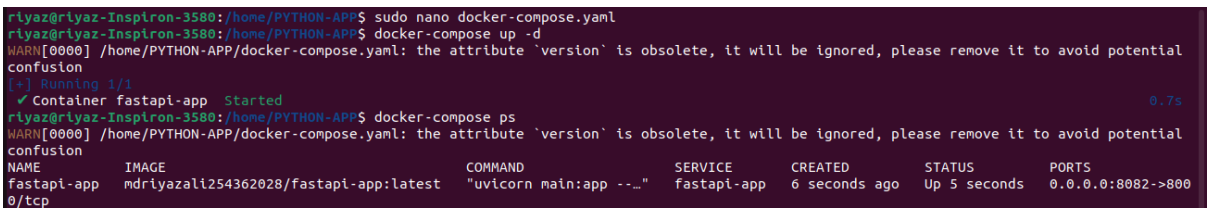
Solutions

- Configured Docker's internal network for seamless inter-container communication.
- Mounted a persistent volume to retain the database state.

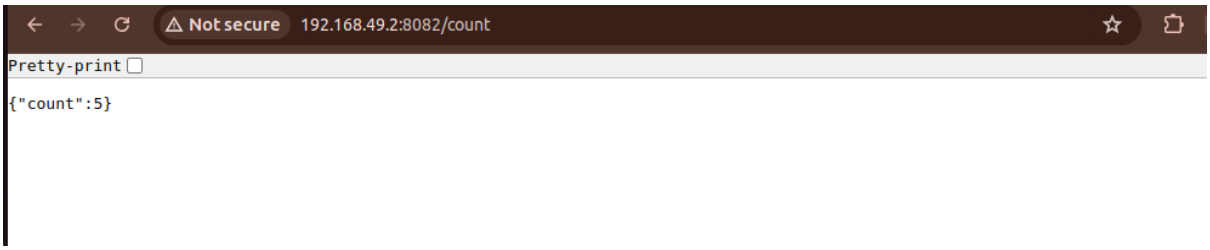
Docker-compose file:

```yaml
version: "3.8"
services:
  fastapi-app:
    image: mdriyazali254362028/fastapi-app:latest
    container_name: fastapi-app
    ports:
      - "8082:8000"
    restart: always
```

Running the docker-compose



Accessing the application with docker-compose up:



---

4. AWS Deployment

Services Used

- **Amazon Route 53**: Domain management and routing.
- **Elastic Load Balancer (ELB)**: Distributes traffic to Kubernetes pods.
- **Amazon EKS**: Orchestrates containerized applications.
- **Amazon RDS**: Provides a managed database for persistent storage.
- **Amazon S3**: Optional logging and static asset storage.
- **CloudWatch**: Monitors logs and system metrics.
- **AWS IAM**: Secures inter-service communication.

Key Considerations

- Implement auto-scaling for Kubernetes pods to handle fluctuating traffic.
- Use multi-AZ RDS deployments for high availability.

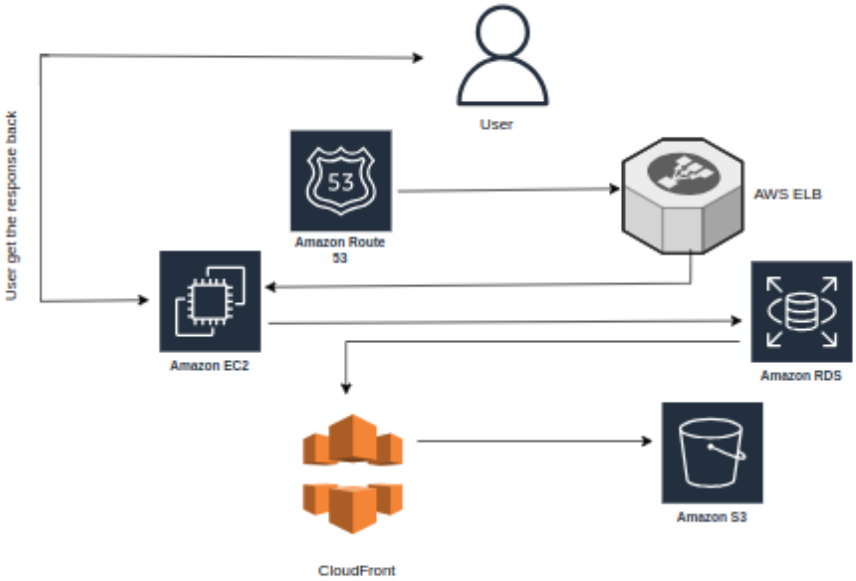
Challenges

- Configuring IAM roles with least privilege access.
- Balancing cost-effectiveness with system performance.

Solutions

- Created IAM policies specific to each service.
- Optimized Kubernetes auto-scaling configurations to minimize idle resources.

Architecture Diagram:



How the Architecture Works

This architecture is designed to deliver a highly available, secure, and scalable application using AWS services. The user interacts with the system by sending requests through Amazon Route 53, which acts as a DNS service, routing traffic to the appropriate endpoint. The requests are then routed to CloudFront, a content delivery network (CDN) that caches and serves static and dynamic content closer to the user from its edge locations. This reduces latency and improves user experience.

Dynamic requests are forwarded to the Elastic Load Balancer (ELB), which distributes traffic across multiple Amazon EC2 instances hosting the application. These EC2 instances process the business logic and interact with Amazon RDS, a managed relational database, to fetch or update data. Meanwhile, static assets such as images, JavaScript files, and other resources are stored in Amazon S3 and delivered to the user via CloudFront for faster access. Once the request is processed, the response flows back through the same path, ensuring an optimised delivery experience.

Cost Optimization

The architecture minimizes costs through intelligent resource allocation and service usage. CloudFront significantly reduces costs by caching frequently accessed content at edge locations, thereby lowering the load on the EC2 instances and reducing data transfer costs from the origin. Amazon S3 further lowers expenses by providing a cost-effective storage solution for static assets, which is cheaper than hosting such resources on EC2. Elastic Load Balancer (ELB) ensures that traffic is efficiently distributed across EC2 instances, and coupled with Auto Scaling, it adjusts the number of running instances based on traffic demand. This eliminates the cost of maintaining idle resources during low-traffic periods. Additionally, EC2 instances can be optimized with cost-efficient instance types, such as spot or reserved instances, further reducing operational expenses.

Security

The architecture incorporates several layers of security. Route 53 ensures secure and highly available DNS routing, while CloudFront enhances security by integrating with AWS Shield to protect against DDoS attacks. Elastic Load Balancer adds resilience by isolating the application from direct user access and ensuring encrypted traffic with SSL/TLS termination. Virtual Private Cloud (VPC) ensures that EC2 and RDS instances are deployed in private subnets, isolating them from public access and enabling secure communication. Additionally, IAM roles and policies can be used to enforce fine-grained access control, ensuring that only authorized users and services can interact with the infrastructure.

5. Kubernetes Deployment

Deployment Strategy

Kubernetes manifests were created to define application pods, services, and networking rules. Helm charts were used for streamlined deployments.

Key Considerations

- Use a Deployment object for pod management.
- Configure a NodePort service for local testing, and ClusterIP or LoadBalancer for production.

Challenges

- YAML syntax errors during configuration.
- Ensuring pod restarts without data loss.

Solutions

- Validated YAML using `kubectl` commands.
- Configured Kubernetes Persistent Volumes for database storage.

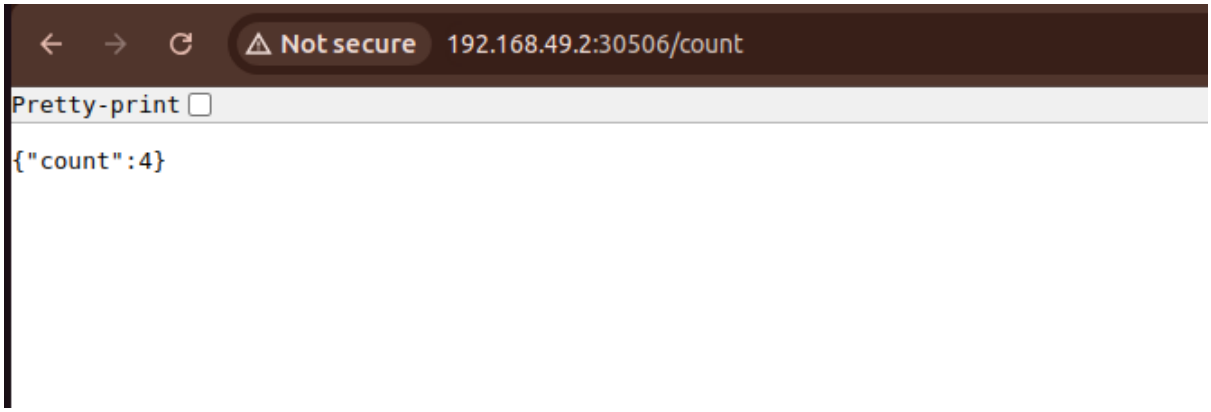
Deployment file:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: fastapi-app
spec:
  replicas: 1
  selector:
    matchLabels:
      app: fastapi
  template:
    metadata:
      labels:
        app: fastapi
    spec:
      containers:
      - name: fastapi-app
        image: mdriyazali254362028/fastapi-app:latest # Use the Docker Hub image
        ports:
        - containerPort: 8000
```

Getting pods and service:

```
riyaz@riyaz-Inspiron-3580:/home/PYTHON-APP$ kubectl get pods
NAME                              READY   STATUS             RESTARTS        AGE
fastapi-app-5d655d9967-stfqz      1/1     Running            0               14s
nginx-deployment-576c6b7b6-shd54  1/1     Running            2 (103m ago)    52d
stress-test                       0/1     CrashLoopBackOff   136 (35s ago)   74d
riyaz@riyaz-Inspiron-3580:/home/PYTHON-APP$ kubectl apply -f service.yaml
service/fastapi-service unchanged
riyaz@riyaz-Inspiron-3580:/home/PYTHON-APP$ cat service.yaml
apiVersion: v1
kind: Service
metadata:
  name: fastapi-service
spec:
  type: LoadBalancer
  ports:
  - port: 80               # Exposed port for external access
    targetPort: 8000       # Port on which the FastAPI app is running in the container
  selector:
    app: fastapi           # Matches the label in the deployment
riyaz@riyaz-Inspiron-3580:/home/PYTHON-APP$ kubectl get services
NAME              TYPE           CLUSTER-IP       EXTERNAL-IP   PORT(S)        AGE
fastapi-service   LoadBalancer   10.111.37.39     <pending>     80:30506/TCP   103m
kubernetes        ClusterIP      10.96.0.1        <none>        443/TCP        74d
nginx-service     NodePort       10.108.218.194   <none>        80:30000/TCP   52d
riyaz@riyaz-Inspiron-3580:/home/PYTHON-APP$ minikube service fastapi-service --url
http://192.168.49.2:30506
riyaz@riyaz-Inspiron-3580:/home/PYTHON-APP$
```

Accessing in browser:



Helm Chart for FastAPI Application

This guide outlines the creation and deployment of a Helm chart for a FastAPI application. Helm simplifies Kubernetes deployments with parameterized templates, enabling seamless configuration, upgrades, and rollbacks.

Helm Chart Components

1. Chart Metadata (Chart.yaml)

Defines essential metadata:

- Name: Application/service (e.g., fastapi-app).
- Version: Helm chart version for tracking.
- App Version: Deployed application version (matches container tags).

2. Configuration File (values.yaml)

Centralized default configurations for deployment:

- Replica Count: Number of application pods
- Image Info: Docker image repository and tag.
- Service Type: Expose via LoadBalancer, NodePort, etc.
- Ports: Internal/external access points.
- Resources: CPU/memory allocation.

3. Templates (templates/)

Parameterized Kubernetes manifests:

- Deployment: Configures pods, replicas, container images, ports, and resource limits.
- Service: Defines network exposure with service type, ports, and pod selectors.
- Templates use placeholders (e.g., {{ .Values.replicaCount }}) to dynamically inject configurations from values.yaml.

Deployment Workflow

Package the Chart: Create a .tgz file.

- Install the Chart: Render templates and apply them to the cluster.
- Verify Deployment: Confirm resources (pods, services, deployments).
- Upgrade/Rollback: Helm tracks releases, allowing seamless updates or reverts**.**

Customization

Use values.yaml for staging (low resources) or production (scalable, resource-intensive). Override with environment-specific files (e.g., production-values.yaml).

Benefits of Helm

- Consistency: Uniform deployments.
- Flexibility: Adapt configurations easily.
- Efficiency: Simplifies deployment processes.
- Version Control: Tracks changes and enables rollbacks.
- Reusability: Easily adaptable for different environments**.**

Conclusion

A Helm chart for FastAPI ensures scalable, reliable, and maintainable Kubernetes deployments with minimal effort.

.

6. Monitoring and Optimization

CloudWatch Integration

CloudWatch was configured to monitor key metrics:

- Pod CPU and memory usage.
- RDS query performance and storage usage.

Key Considerations

- Set up alerts for high CPU/memory usage.
- Analyze logs for application errors.

Challenges & Lessons Learned

Challenges

1. **Security**: Configuring IAM roles for secure service interactions.
2. **Performance**: Managing resource scaling during high traffic.
3. **Cost Control**: Avoiding over-provisioning of AWS resources.

Lessons Learned

- Modular configuration files simplify testing and troubleshooting.
- Regularly test deployments on staging environments to identify issues early.
- Automate repetitive tasks with CI/CD pipelines for efficiency.

Future Improvements

1. **CI/CD Integration**: Automate deployments to reduce manual intervention.
2. **Enhanced Monitoring**: Incorporate Prometheus and Grafana for detailed observability.
3. **Cost Analysis**: Use AWS Cost Explorer to optimize resource usage and reduce costs.

Conclusion

This document provides a structured approach to deploying a FastAPI application with scalability, security, and efficiency in mind. By addressing challenges and leveraging AWS and Kubernetes capabilities, the deployment achieves production-grade reliability.
