Robert Gordon University

Cloud Computing – CMM707

Course Work Report

MSc in Big Data Analytics

Semester 1

Submitted by:

S.Z. Raeesul Islam

20232953 / 2409649

GitHub Link: https://github.com/Raeesul25/MediTrack_HealthSync

Table of Contents

Contents

1. System Architecture – MediTrack

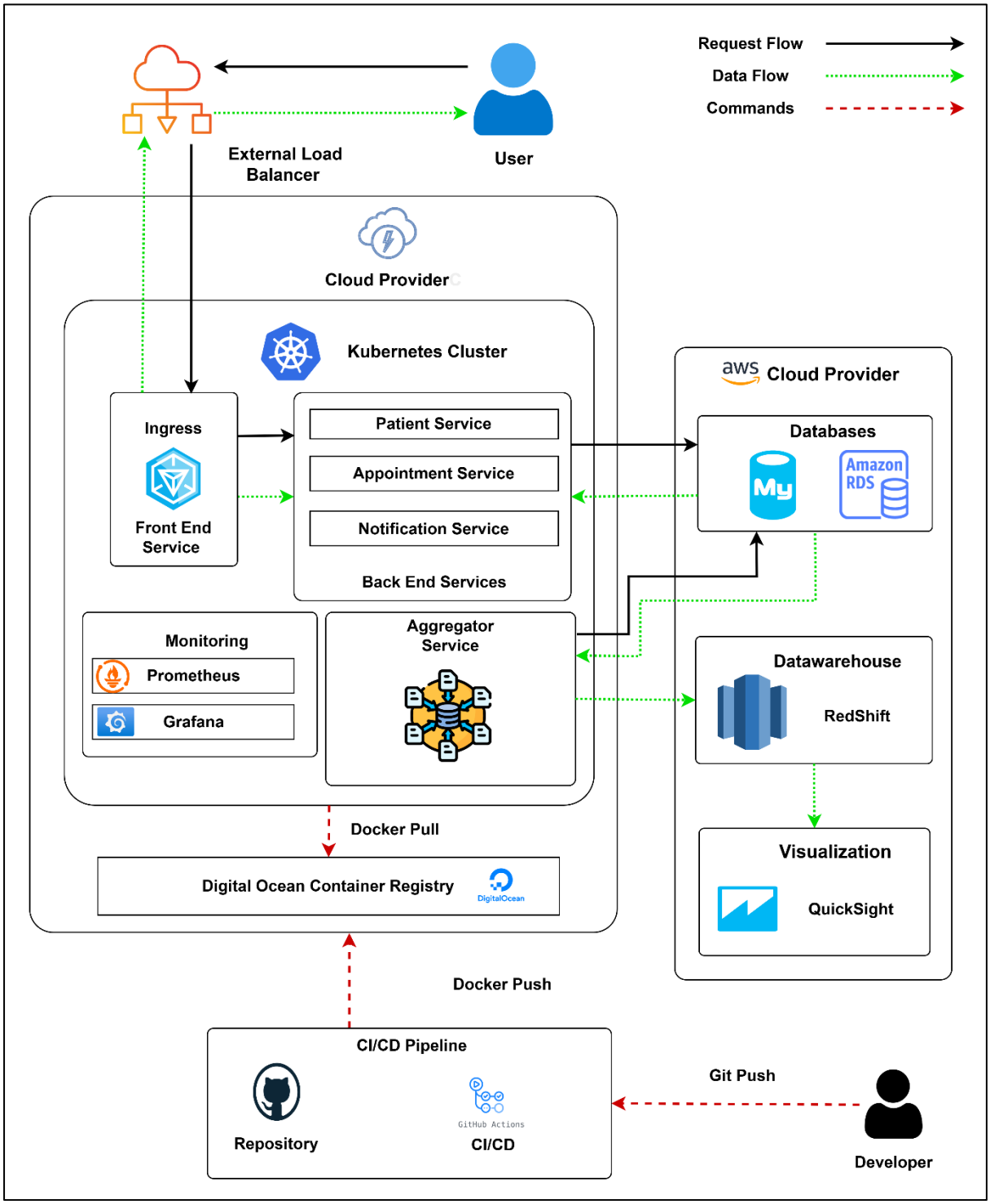

Figure 1. Solution Architecture Diagram

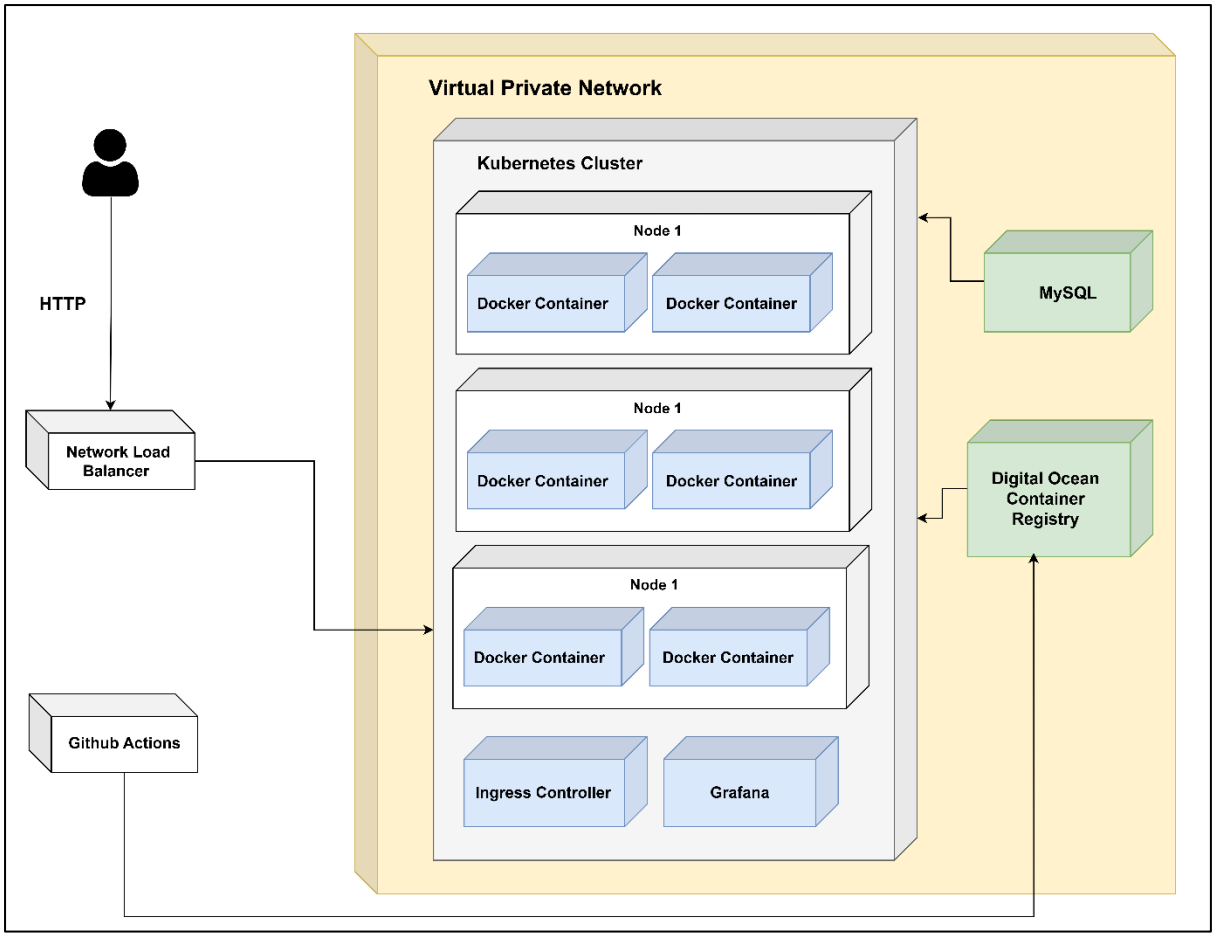

Figure 2. Deployment Diagram

MediTrack is a platform that combines **scalability, security, fault tolerance, and affordability** by utilizing **DigitalOcean** for **Docker image** storage and **Kubernetes** deployment management. The platform uses Kubernetes' **auto-scaling** capabilities, a network load balancer, and VPN for service isolation and **secure access** to Docker images. It also uses role-based access control and self-healing mechanisms for **fault tolerance and high availability. Real-time monitoring** with Prometheus and Grafana enables quick issue resolution. DigitalOcean's **cost-efficient** Kubernetes services and pay-as-you-go pricing model reduce operational expenses. Open-source tools for monitoring and automation minimize licensing costs. The CI/CD pipeline streamlines deployments, reducing manual effort and accelerating delivery. This comprehensive solution ensures MediTrack remains scalable, secure, reliable, and cost-effective while meeting the demands of a growing user base.

1.1. The Solution’s Internal Communication

The MediTrack solution is designed for seamless interaction between **users, services, and backend systems**. It uses an **external load balancer** to **route HTTP requests** to the **Kubernetes cluster**, with the **ingress controller** directing traffic to appropriate services. These services communicate internally using service discovery and **DNS** provided by Kubernetes, ensuring **efficient data flow**. **Data requests** are forwarded

to databases hosted on DigitalOcean for **fast and reliable** storage. The **Aggregator Service** consolidates data for complex operations, and the **CI/CD pipeline** automates communication between the developer's repository and the DigitalOcean Container Registry. **Monitoring tools** like Prometheus and Grafana continuously communicate with the cluster, **enabling real-time** issue detection. This layered and modular communication architecture ensures **efficient request routing, secure data flow, and reliable system operations** across the MediTrack platform.

2. Security and Ethics Challenges

The MediTrack platform faces military security and ethical dilemmas that must be addressed to enhance the platform's credibility and integrity of healthcare delivery. On the **security** front, given that the platform handles sensitive patient data, it is potentially able to face **cyberattacks** such as breaches, ransomware, or unauthorized access. **Patient privacy** is of foremost importance and ensuring compliance with the **Health Insurance Portability and Accountability Act (HIPAA)** or other similar regulations is vital. Other weaknesses may lie in **data transmission across microservices**, the cloud infrastructure, and external systems. **Securing APIs, encrypting data** both at rest and in transit, and **implementing multi-factor authentication** are some of the measures used to **mitigate** these incidences. Moreover, insider **threats and misuse** of employees' access privileges present serious risks to data integrity but may be mitigated through **Role-Based Access Control (RBAC)** and activity monitoring.

Ethical difficulties begin with data ownership, consent, and utilization. Patients need to have some visibility for their control over the data; information should be used for analytics or research with explicit consent from the patients. In practice, the platform's algorithm could indeed be biased or advantage some group type over another in preference for service. This can create further inequalities for some groups in terms of fairness in AI-driven analytics and decision-making processes.

There's a great concern on whether patient insights gained from aggregated data are used ethically. On one hand, data analytics is expected to confer operational efficiency while improving patient outcomes. However, if these results are misused for the commercial gain of a company without the consent of the patient, it may create an ethical breach. The platform should ensure accountability and transparency about data collection, processing, and utilization.

The joint approach to the above challenges must emphasize the formulation of robust security protocols, ethical governance frameworks, application of regular audits, and compliance with healthcare data protection laws. Such proactive risk mitigation would see the MediTrack platform able to foster trust and build on very high standards of security and ethics.

3. CI/CD Process

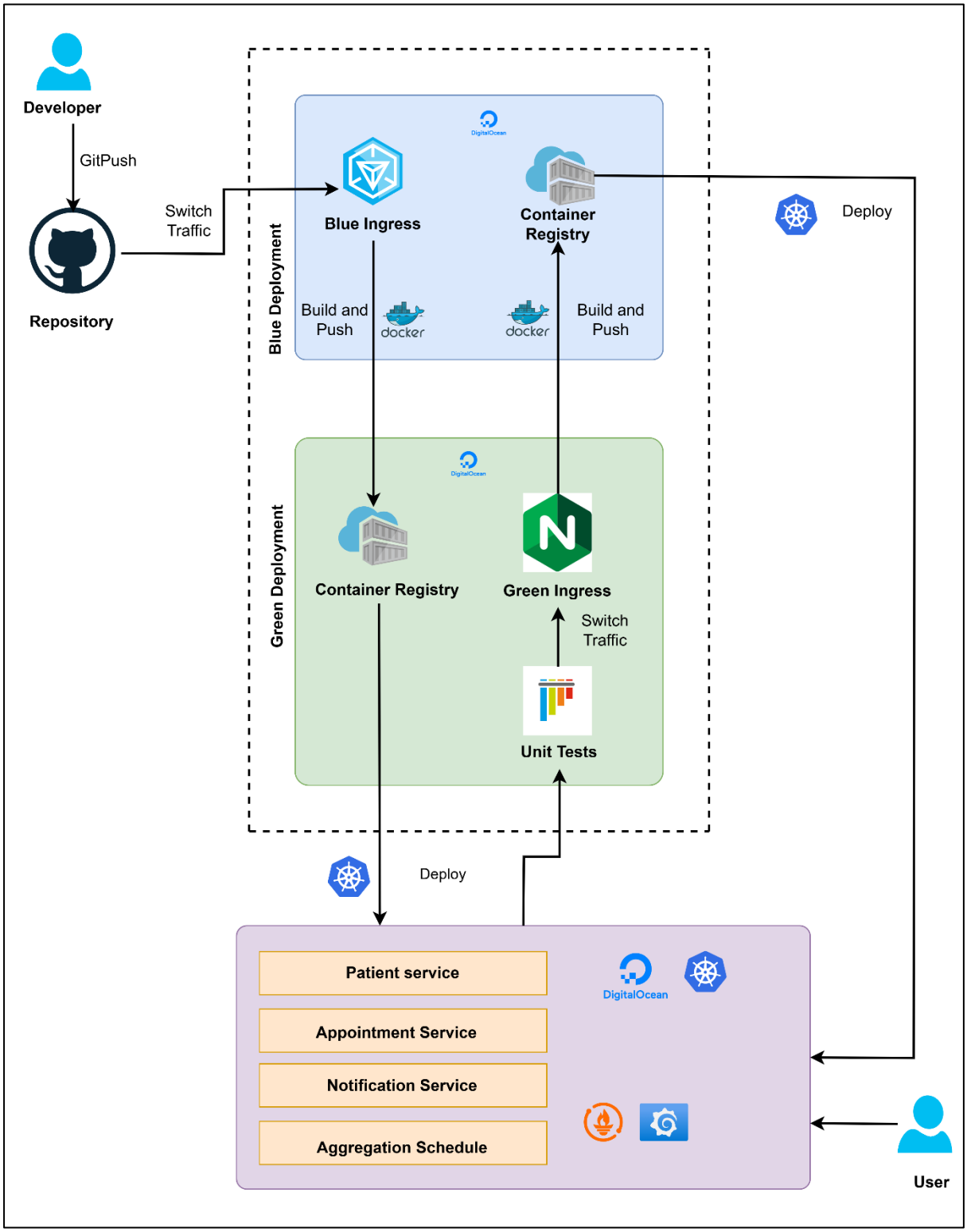

Figure 3. CI/CD Process Diagram

The MediTrack platform's **CI/CD process** is automated, ensuring seamless and reliable deployment through **GitHub Actions workflows.** The pipeline pulls the latest code, creates **Docker images** for **microservices**, and pushes them to the **DigitalOcean Container Registry.** The containers are deployed to the **Green Environment** within

the Kubernetes cluster, while production traffic continues through the **Blue Environment.** Automated unit tests are executed in the **Green Environment,** and if successful, the pipeline updates the **Blue Environment.** The **Ingress Controller** switches production traffic to the **Green Environment,** making it active. Post-deployment testing using tools like Postman ensures stability. If a failure occurs during **Green deployment,** the pipeline halts, notifies developers, and continues routing traffic to the stable Blue Environment, providing a robust fallback mechanism. This process ensures efficient, secure, and resilient deployments.

Security Challenges

The MediTrack platform faces security challenges, including **data breaches, unauthorized access** due to Kubernetes misconfigurations, and secrets management. TLS encryption is used for transit data and AES-256 for rest data. **Kubernetes misconfigurations** are addressed through namespace isolation and network policies. Secrets are securely stored using GitHub Secrets and tools like Vault. Production-grade security measures are applied to both Blue and Green environments to prevent exploitation during testing. CI/CD pipeline integrity is protected through signed commits, secure GitHub Actions runners, and continuous security monitoring to prevent malicious deployments and ensure pipeline security.

Ethical Challenges

The platform addresses ethical challenges to maintain trust and compliance, including patient consent, transparency with healthcare providers, data ownership disputes, algorithm bias, and environmental impact. Clear communication of data policies and explicit consent are crucial for patients to understand their data processing. Sharing deployment logs and testing results with stakeholders helps resolve these issues. Data ownership disputes are addressed by ensuring patients retain ownership of their health records. Regular audits of algorithms and datasets ensure accuracy and fairness. The platform also addresses environmental concerns by optimizing pipeline efficiency and using green-certified cloud providers.

4. Implementation

4.1. Microservices Implementation

We start by setting up the AWS RDS instance, creating the database, and connecting it to MySQL Workbench.

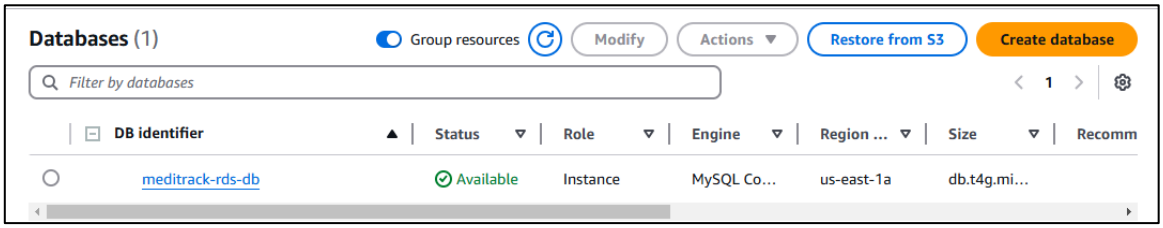

Figure 4. Amazon RDS Instance

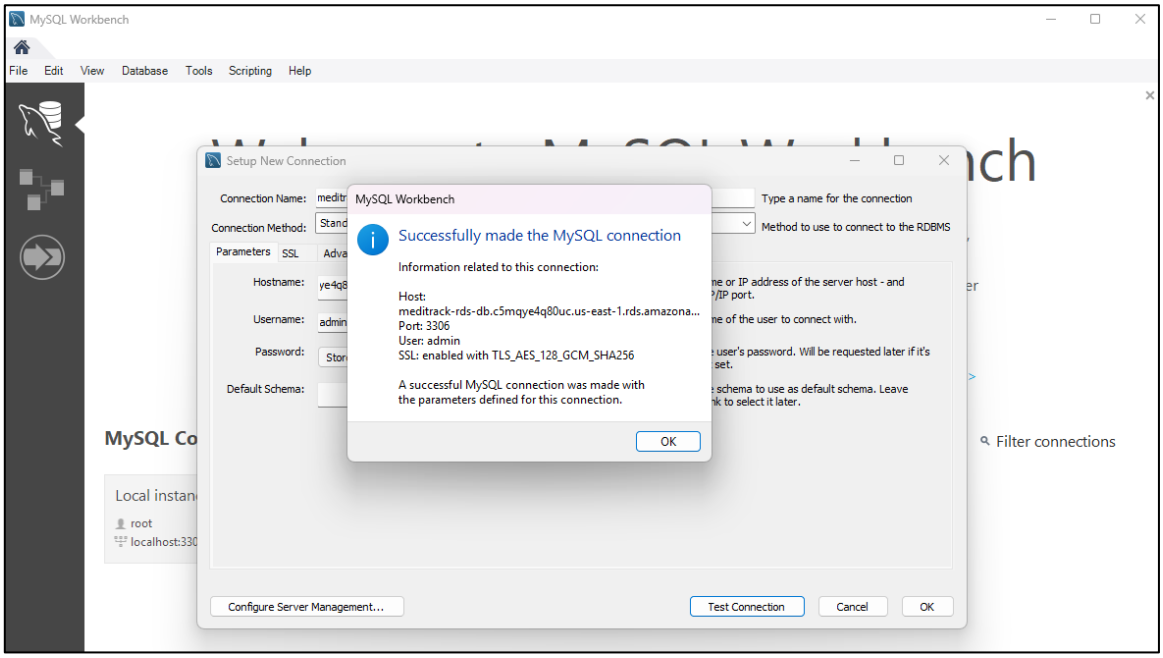

Figure 5. Amazon RDS instance connected to MySQL

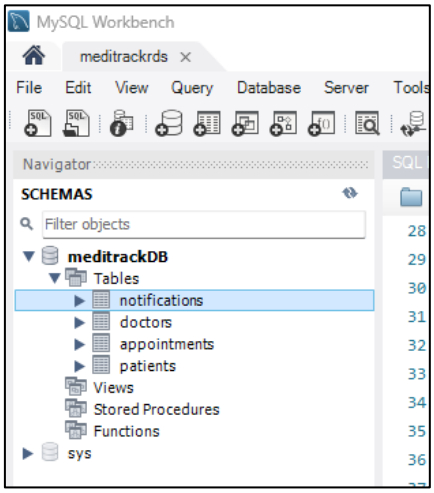

Figure 6. MediTrack DB tables

The first step is to develop and test the code on a **local machine** to ensure it works correctly. Once verified, the code is **pushed to GitHub using Git**, enabling version control, collaboration, and centralized storage.

```python
from flask import Flask, request, jsonify, render_template
import mysql.connector

app = Flask(__name__)

# Database connection configuration
def get_db_connection():
    return mysql.connector.connect(
        host="meditrack-rds-db.c5mqye4q80uc.us-east-1.rds.amazonaws.com",
        user="admin",
        password="MeditrackDB",
        database="meditrackDB"
    )

# Route for the front-end page
@app.route('/')
def index():
    return render_template('index.html')

# API route to add a new patient
@app.route('/api/patients', methods=['POST'])
def add_patient():
    data = request.json
    name = data.get('name')
    age = data.get('age')
    gender = data.get('gender')
    contact_number = data.get('contact_number')
    email = data.get('email')
    medical_history = data.get('medical_history')
```

Figure 7. Patient Record Service Code

```python
from flask import Flask, request, jsonify, render_template
import mysql.connector

app = Flask(__name__)

# Database connection configuration
def get_db_connection():
    return mysql.connector.connect(
        host="meditrack-rds-db.c5mqye4q80uc.us-east-1.rds.amazonaws.com",
        user="admin",
        password="MeditrackDB",
        database="meditrackDB"
    )

# Route for the front-end page
@app.route('/')
def index():
    return render_template('index.html')

# API to get all available doctors
@app.route('/api/doctors', methods=['GET'])
def get_doctors():
    try:
        connection = get_db_connection()
        cursor = connection.cursor(dictionary=True)

        query = "SELECT * FROM doctors"
        cursor.execute(query)
```

Figure 8. Appointment Scheduling Service Code

```python
from flask import Flask, request, jsonify, render_template
import mysql.connector
from datetime import datetime

app = Flask(__name__)

# Database connection configuration
def get_db_connection():
    return mysql.connector.connect(
        host="meditrack-rds-db.c5mqye4q80uc.us-east-1.rds.amazonaws.com",
        user="admin",
        password="MeditrackDB",
        database="meditrackDB"
    )

# Route for the front-end page
@app.route('/')
def index():
    return render_template('index.html')

# API to send a notification
@app.route('/api/notifications', methods=['POST'])
def send_notification():
    data = request.json
    patient_id = data.get('patient_id')
    message = data.get('message')

    try:
```

Figure 9. Notification Service Code

4.2. Aggregator Service Implementation

Created the AWS Redshift serverless namespace/workgroup

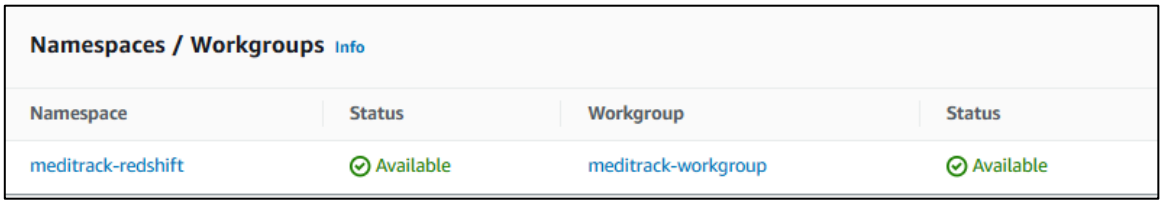

Figure 10. Redshift Namespace/Workgroup

Create Datawarehouse

```sql
CREATE TABLE total_appointments_per_doctor (
    doctor_name VARCHAR(255) NOT NULL,
    total_appointments INT NOT NULL,
    PRIMARY KEY (doctor_name)
);

CREATE TABLE appointments_per_month_per_doctor (
    month_year VARCHAR(50) NOT NULL,
    doctor_name VARCHAR(255) NOT NULL,
    appointment_count INT NOT NULL,
    PRIMARY KEY (month_year, doctor_name)
);

CREATE TABLE doctors_per_speciality (
    specialty VARCHAR(255) NOT NULL,
    doctor_count INT NOT NULL,
    PRIMARY KEY (specialty)
);
```

Figure 11. Redshift Datawarehouse

Develop and test the code on a **local machine** to ensure it works correctly. Once verified, the code is **pushed to GitHub using Git**.

```python
# function for aggregate total appointments
def aggregate_total_appointments():
    try:
        # Fetch all appointments from MySQL
        connection = get_db_connection()
        cursor = connection.cursor()
        query = "SELECT * FROM appointments"

        # Aggregate total appointments per doctor
        appointments_df = pd.read_sql(query, connection)
        total_appointments_per_doctor = appointments_df.groupby('doctor_id').size().reset_index(name='total_appointments')

        # Fetch doctor names from PostgreSQL
        doctor_ids = total_appointments_per_doctor['doctor_id'].tolist()
        doctor_query = f"SELECT doctor_id, name FROM doctors WHERE doctor_id IN ({','.join(map(str, doctor_ids))})"
        cursor.execute(doctor_query)
        doctors_data = cursor.fetchall()
        doctors_df = pd.DataFrame(doctors_data, columns=['doctor_id', 'doctor_name'])

        appointments_df['doctor_id'] = appointments_df['doctor_id'].astype(int)
        doctors_df['doctor_id'] = doctors_df['doctor_id'].astype(int)
        total_appointments_per_doctor['doctor_id'] = total_appointments_per_doctor['doctor_id'].astype(int)

        # Merge with doctor names
        merged_df = total_appointments_per_doctor.merge(doctors_df, on='doctor_id', how='left')
        test = pd.DataFrame(merged_df , columns=['doctor_name', 'total_appointments'])

        return test

    except Exception as e:
        print("Error connecting to the database:", e)
```

Figure 12. Aggregate total appointments

```python
# function for aggregate doctors per specialty
def aggregate_doctors_per_specialty():
    try:
        # Connect to MySQL
        connection = get_db_connection()
        cursor = connection.cursor()

        doctor_query = "SELECT specialty, COUNT(*) FROM doctors GROUP BY specialty"
        cursor.execute(doctor_query)
        specialty_data = cursor.fetchall()

        specialty_df = pd.DataFrame(specialty_data, columns=['specialty', 'doctor_count'])

        return specialty_df

    except Exception as e:
        print("Error connecting to the database:", e)
```

Figure 13. Aggregate doctors per specialty

```python
# function for aggregate appointments per month
def aggregate_appointments_per_month():

    try:
        # Fetch all appointments from MySQL
        connection = get_db_connection()
        cursor = connection.cursor()
        query = "SELECT * FROM appointments"

        # Convert into dataframe
        appointments_df = pd.read_sql(query, connection)

        # Convert appointment date to datetime
        appointments_df['appointment_date'] = pd.to_datetime(appointments_df['appointment_date'])

        # Extract month-year and doctor name
        appointments_df['month_year'] = appointments_df['appointment_date'].dt.strftime('%B %Y')

        # Group by month_year and doctor
        month_appointments = appointments_df.groupby(['month_year', 'doctor_id']).size().reset_index(name='appointment_count')

        # Fetch doctor names from PostgreSQL
        doctor_ids = month_appointments['doctor_id'].tolist()
        doctor_query = f"SELECT doctor_id, name FROM doctors WHERE doctor_id IN ({','.join(map(str, doctor_ids))})"
        cursor.execute(doctor_query)
        doctors_data = cursor.fetchall()
        doctors_df = pd.DataFrame(doctors_data, columns=['doctor_id', 'doctor_name'])

        doctors_df['doctor_id'] = doctors_df['doctor_id'].astype(int)
        month_appointments['doctor_id'] = month_appointments['doctor_id'].astype(int)

        # Merge with doctor names
        merged_df = month_appointments.merge(doctors_df, on='doctor_id', how='left')

        return merged_df[['month_year', 'doctor_name', 'appointment_count']]
```

Figure 14. Aggregate appointments per month

```python
# function for insert Total appointments per doctors
def insert_total_appointments_per_doctor(aggregated_df):

    try:
        conn_rs = psycopg2.connect(
            host = RS_HOST,
            port = RS_PORT,
            dbname = RS_DBNAME,
            user = RS_USER,
            password = RS_PASSWORD
        )
        cur_rs  = conn_rs.cursor()

        # Iterate through the DataFrame and insert/update records in Redshift
        for index, row in aggregated_df.iterrows():
            doctor_name = row['doctor_name']
            total_appointments = row['total_appointments']

            # Step 1: Update existing records
            update_query = """
                UPDATE total_appointments_per_doctor
                SET total_appointments = %s
                WHERE doctor_name = %s;
            """
            cur_rs.execute(update_query, (total_appointments, doctor_name))

            # Step 2: If no records were updated, insert the new record
            insert_query = """
                INSERT INTO total_appointments_per_doctor (doctor_name, total_appointments)
                SELECT %s, %s
                WHERE NOT EXISTS (
                    SELECT 1 FROM total_appointments_per_doctor WHERE doctor_name = %s
                );
            """
            cur_rs.execute(insert_query, (doctor_name, total_appointments, doctor_name))
```

Figure 15. Insert Total appointments per doctors

```python
# function for insert appointments per month for per doctor
def insert_appointments_per_month_per_doctor(aggregated_df):

    try:
        conn_rs = psycopg2.connect(
            host = RS_HOST,
            port = RS_PORT,
            dbname = RS_DBNAME,
            user = RS_USER,
            password = RS_PASSWORD
        )

        cur_rs  = conn_rs.cursor()

        # Iterate through the DataFrame and insert/update records in Redshift
        for index, row in aggregated_df.iterrows():
            month_year = row['month_year']
            doctor_name = row['doctor_name']
            total_appointments = row['appointment_count']

            # Step 1: Update existing records
            update_query = """
                UPDATE appointments_per_month_per_doctor
                SET appointment_count  = %s
                WHERE month_year = %s AND doctor_name = %s;
            """
            cur_rs.execute(update_query, (total_appointments, month_year, doctor_name))

            # Step 2: If no records were updated, insert the new record
            insert_query = """
                INSERT INTO appointments_per_month_per_doctor (month_year, doctor_name, appointment_count )
                SELECT %s, %s, %s
                WHERE NOT EXISTS (
                    SELECT 1 FROM appointments_per_month_per_doctor
                    WHERE month_year = %s AND doctor_name = %s
```

Figure 16. Insert appointments per month for per doctor

```python
# function for insert doctors per specialty
def insert_doctors_per_speciality(doctors_per_speciality_df):
    # Assuming you already have a connection to Redshift
    try:
        conn_rs = psycopg2.connect(
            host = RS_HOST,
            port = RS_PORT,
            dbname = RS_DBNAME,
            user = RS_USER,
            password = RS_PASSWORD
        )

        cur_rs  = conn_rs.cursor()

        # Iterate through the DataFrame and insert/update records in Redshift
        for index, row in doctors_per_speciality_df.iterrows():
            specialty = row["specialty"]
            doctor_count = row["doctor_count"]

            # Use a Redshift-compatible UPSERT (MERGE) query
            merge_query = f"""
                BEGIN;

                DELETE FROM doctors_per_speciality
                WHERE specialty = %s;

                INSERT INTO doctors_per_speciality (specialty, doctor_count)
                VALUES (%s, %s);

                COMMIT;
            """
            cur_rs.execute(merge_query, (specialty, specialty, doctor_count))

        # Commit and close the connection
        conn_rs.commit()
```

Figure 17. Insert doctors per specialty

4.3. CI/CD Process Implementation

```
# Copy application code and dependencies
COPY . .

# Install dependencies
RUN pip install -r requirements.txt

# Copy test files
COPY templates ./templates
COPY tests ./tests

# Expose the application port
EXPOSE 5000

# Run the application
CMD ["python", "patient_app.py"]
```

Figure 18. Dockerfile

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: patient-record-service-blue
spec:
  replicas: 1
  selector:
    matchLabels:
      app: patient-record-service
      version: blue
  template:
    metadata:
      labels:
        app: patient-record-service
        version: blue

    spec:
      containers:
      - name: patient-record-service
        image: registry.digitalocean.com/meditrackcontainer/patient-record:blue
        ports:
        - containerPort: 80
      imagePullSecrets:
      - name: do-secret

---
apiVersion: v1
kind: Service
metadata:
  name: patient-record-service-blue
spec:
  selector:
    app: patient-record-service
    version: blue
  ports:
```

Figure 19. Blue Service and Deployment

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: patient-record-service-green
spec:
  replicas: 1
  selector:
    matchLabels:
      app: patient-record-service
      version: green
  template:
    metadata:
      labels:
        app: patient-record-service
        version: green

    spec:
      containers:
      - name: patient-record-service
        image: registry.digitalocean.com/meditrackcontainer/patient-record:green
        ports:
        - containerPort: 80
      imagePullSecrets:
      - name: do-secret

---
apiVersion: v1
kind: Service
metadata:
  name: patient-record-service-green
spec:
  selector:
    app: patient-record-service
    version: green
  ports:
```

Figure 20. Green service and deployment

```yaml
apiVersion: batch/v1
kind: CronJob
metadata:
  name: aggregator-cronjob
spec:
  schedule: "30 18 * * *" # Run at 1 AM daily
  jobTemplate:
    spec:
      template:
        spec:
          containers:
          - name: aggregator
            image: registry.digitalocean.com/meditrackcontainer/aggregate-service:latest
            imagePullPolicy: Always

          imagePullSecrets:
          - name: do-secret
          restartPolicy: Never
```

Figure 21. Aggregator cronjob YAML

ingress-blue.yaml

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: microservices-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target:
spec:
  ingressClassName: nginx
  rules:
  - host: 137.184.249.159.nip.io
    http:
      paths:
      - path: /get_patients
        pathType: Prefix
        backend:
          service:
            name: patient-record-service-blue
            port:
              number: 80
      - path: /add_patient
        pathType: Exact
        backend:
          service:
            name: patient-record-service-blue
            port:
              number: 80
      - path: /update_patient
        pathType: Prefix
        backend:
          service:
            name: patient-record-service-blue
```

Figure 22. Blue ingress YAML

ingress-green.yaml

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: microservices-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target:
spec:
  ingressClassName: nginx
  rules:
  - host: 137.184.249.159.nip.io
    http:
      paths:
      - path: /get_patients
        pathType: Prefix
        backend:
          service:
            name: patient-record-service-green
            port:
              number: 80
      - path: /add_patient
        pathType: Exact
        backend:
          service:
            name: patient-record-service-green
            port:
              number: 80
      - path: /update_patient
        pathType: Prefix
        backend:
          service:
            name: patient-record-service-green
            port:
```

Figure 23. Green ingress YAML

4.3.1. CI/CD Deployment

The CI/CD pipeline begins when developers push changes to the main branch of the Git repository. This triggers an automated workflow, such as a GitHub Actions pipeline, which initiates the process of building, testing, and deploying the application.

```yaml
on:
  push:
    branches:
      - main
```

Figure 24. Triggering based on a git push

To ensure zero downtime during deployment, the pipeline first directs all user traffic to the Blue Environment via the Blue Ingress Controller. This isolates the Green Environment, allowing it to be used for testing and updates without impacting active users.

```yaml
jobs:
  change-to-blue:
    steps:

      # Log in to DigitalOcean container registry
      - name: Log in to DigitalOcean
        run: echo "${{ secrets.DIGITALOCEAN_ACCESS_TOKEN }}" | docker login registry.digitalocean.com -u doctl --password-stdin

      - name: Build and push aggregator
        run: |
          docker build -t registry.digitalocean.com/meditrackcontainer/aggregate-service:latest ./aggregatorService
          docker push registry.digitalocean.com/meditrackcontainer/aggregate-service:latest

      - name: Checkout code
        uses: actions/checkout@v3

      - name: Set up doctl
        uses: digitalocean/action-doctl@v2
        with:
          token: ${{ secrets.DIGITALOCEAN_ACCESS_TOKEN }}

      - name: Install kubectl
        uses: azure/setup-kubectl@v3
        with:
          version: 'latest'

      - name: Set up kubeconfig
        run: doctl kubernetes cluster kubeconfig save k8s-1-31-1-do-5-sgp1-1734795051852

      - name: Apply Kubernetes Manifests blue
        run: |
          kubectl apply -f ingress-blue.yaml
          kubectl apply -f ./aggregatorService/aggregator-cronjob.yaml
```

Figure 25. Switching traffic to blue ingress

The pipeline pulls the latest application code and begins the build process. Docker images for all relevant microservices are created and tagged with unique versions. These images are then securely pushed to the DigitalOcean Container Registry, ensuring proper version management and storage.

```yaml
jobs:
  build-green:
    needs: change-to-blue
    runs-on: ubuntu-latest

    steps:
      # Checkout the code
      - name: Checkout code
        uses: actions/checkout@v3

      # Log in to DigitalOcean container registry
      - name: Log in to DigitalOcean
        run: echo "${{ secrets.DIGITALOCEAN_ACCESS_TOKEN }}" | docker login registry.digitalocean.com -u doctl --password-stdin

      # Build and push each service
      - name: Build and push Patient Record Service green
        run: |
          docker build -t registry.digitalocean.com/meditrackcontainer/patient-record:green ./patientRecordService
          docker push registry.digitalocean.com/meditrackcontainer/patient-record:green

      - name: Build and push Appointment Scheduling Service green
        run: |
          docker build -t registry.digitalocean.com/meditrackcontainer/appointment-scheduling:green ./appointmentSchedulingService
          docker push registry.digitalocean.com/meditrackcontainer/appointment-scheduling:green

      - name: Build and push Notification Service green
        run: |
          docker build -t registry.digitalocean.com/meditrackcontainer/notification-service:green ./notificationService
          docker push registry.digitalocean.com/meditrackcontainer/notification-service:green
```

Figure 26. Building and pushing green docker images

The Green Environment in the Kubernetes cluster is updated with the newly built Docker images. The Kubernetes manifests are applied, ensuring that the updated services are launched and running within the isolated Green Environment.

```yaml
jobs:
  deploy-green:
    needs: build-green
    runs-on: ubuntu-latest

    steps:
      - name: Checkout code
        uses: actions/checkout@v3

      - name: Set up doctl
        uses: digitalocean/action-doctl@v2
        with:
          token: ${{ secrets.DIGITALOCEAN_ACCESS_TOKEN }}


      - name: Install kubectl
        uses: azure/setup-kubectl@v3
        with:
          version: 'latest'


      - name: Set up kubeconfig
        run: doctl kubernetes cluster kubeconfig save k8s-1-31-1-do-5-sgp1-1734795051852

      - name: Apply Kubernetes Manifests green
        run: |
          kubectl apply -f ./appointmentSchedulingService/appointment-green-deployment.yaml
          kubectl apply -f ./notificationService/notification-green-deployment.yaml
          kubectl apply -f ./patientRecordService/patient-green-deployment.yaml
          kubectl rollout restart deployment patient-record-service-green
          kubectl rollout restart deployment appointment-scheduling-service-green
```

Figure 27. Deploying the Green model

Automated tests, such as unit tests and integration tests, are executed within the Green Environment to validate its stability and functionality.

```yaml
jobs:
  deploy-green:
    steps:
      - name: Port-forward Green Service for Testing Patients
        run: |
          kubectl port-forward service/patient-record-service-green 8080:80 &
          sleep 5  # Wait for port-forward to establish

      - name: Run Tests for patients

        working-directory: ./patientRecordService
        run: |
          python -m venv venv
          source venv/bin/activate
          pip install pytest
          pip install requests
          pytest tests/test_patient.py --maxfail=1 --disable-warnings
          pkill -f 'kubectl port-forward'

      - name: Port-forward Green Service for Testing Doctors
        run: |
          kubectl port-forward service/appointment-scheduling-service-green 8080:80 &
          sleep 5  # Wait for port-forward to establish

      - name: Run Tests for doctors
        working-directory: ./appointmentSchedulingService
        run: |
          python -m venv venv
          source venv/bin/activate
          pip install pytest
          pip install requests
          pytest tests/test_doctor.py --maxfail=1 --disable-warnings
          pkill -f 'kubectl port-forward'
```

Figure 28. Testing green deployment

After successful testing in the Green Environment, the pipeline builds the Green Docker images again, ensuring all changes are accounted for. These images are then pushed to the container registry for the Blue Environment update.

```yaml
jobs:
  build-blue:
    needs: deploy-green
    if: success()  # Only runs if tests pass
    runs-on: ubuntu-latest

    steps:
      # Checkout the code
      - name: Checkout code
        uses: actions/checkout@v3

      # Log in to DigitalOcean container registry
      - name: Log in to DigitalOcean
        run: echo "${{ secrets.DIGITALOCEAN_ACCESS_TOKEN }}" | docker login registry.digitalocean.com -u doctl --password-stdin

      # Build and push each service
      - name: Build and push patient record Service blue
        run: |
          docker build -t registry.digitalocean.com/meditrackcontainer/patient-record:blue ./patientRecordService
          docker push registry.digitalocean.com/meditrackcontainer/patient-record:blue

      - name: Build and push appointment scheduling Service blue
        run: |
          docker build -t registry.digitalocean.com/meditrackcontainer/appointment-scheduling:blue ./appointmentSchedulingService
          docker push registry.digitalocean.com/meditrackcontainer/appointment-scheduling:blue

      - name: Build and push notification Service blue
        run: |
          docker build -t registry.digitalocean.com/meditrackcontainer/notification-service:blue ./notificationService
          docker push registry.digitalocean.com/meditrackcontainer/notification-service:blue
```

Figure 29. Building and pushing green docker images

The Blue Environment is updated with the same stable deployment that passed testing in the Green Environment. Once updated, the Ingress Controller switches traffic to the Green Environment, making it the active deployment for production.

```yaml
jobs:

  deploy:
    needs: build-blue
    runs-on: ubuntu-latest

    steps:
      - name: Checkout code
        uses: actions/checkout@v3

      - name: Set up doctl
        uses: digitalocean/action-doctl@v2
        with:
          token: ${{ secrets.DIGITALOCEAN_ACCESS_TOKEN }}

      - name: Install kubectl
        uses: azure/setup-kubectl@v3
        with:
          version: 'latest'

      - name: Set up kubeconfig
        run: doctl kubernetes cluster kubeconfig save k8s-1-31-1-do-5-sgp1-1734795051852

      - name: Apply Kubernetes Manifests blue
        run: |
          kubectl apply -f ./appointmentSchedulingService/appointment-blue-deployment.yaml
          kubectl rollout restart deployment appointment-scheduling-service-blue
          kubectl apply -f ./notificationService/notification-blue-deployment.yaml
          kubectl apply -f ./patientRecordService/patient-blue-deployment.yaml
          kubectl apply -f ingress-green.yaml
          kubectl rollout restart deployment patient-record-service-blue
          kubectl rollout restart deployment appointment-scheduling-service-blue
```

Figure 30. Deploying blue model and directing traffic to green ingress

After switching traffic, the pipeline verifies the deployment's success through additional tests. If no issues are detected, the pipeline concludes, marking the deployment as successful. In the event of failure, the fallback mechanism ensures traffic remains routed to the stable Blue Environment.

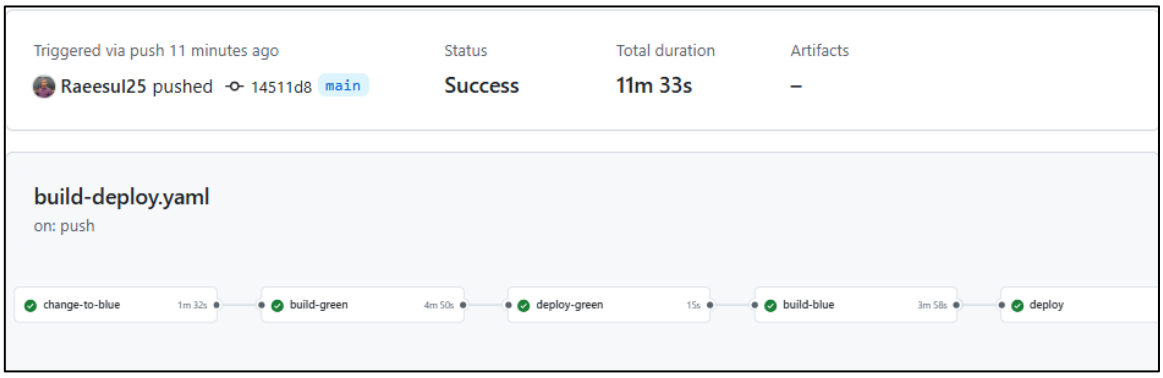

Figure 31. Deployed Successfully

Figure 32. Docker Images

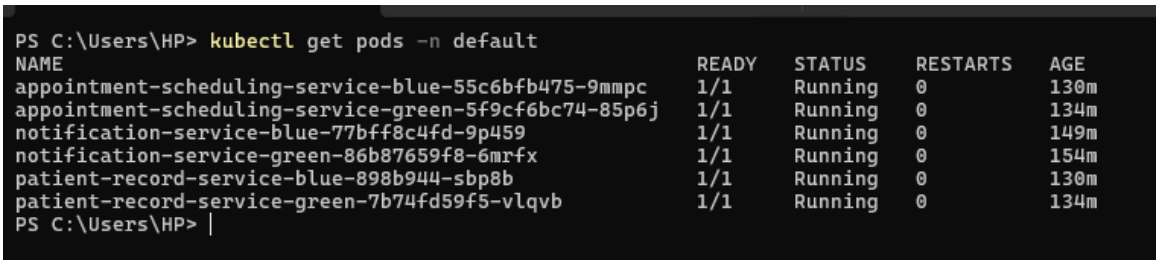

Figure 33. Verify All Pods are in running state

5. Testing the Deployed App

Using API data requests, the Postman application is used to test the deployed microservices.

5.1. Patient Record Service

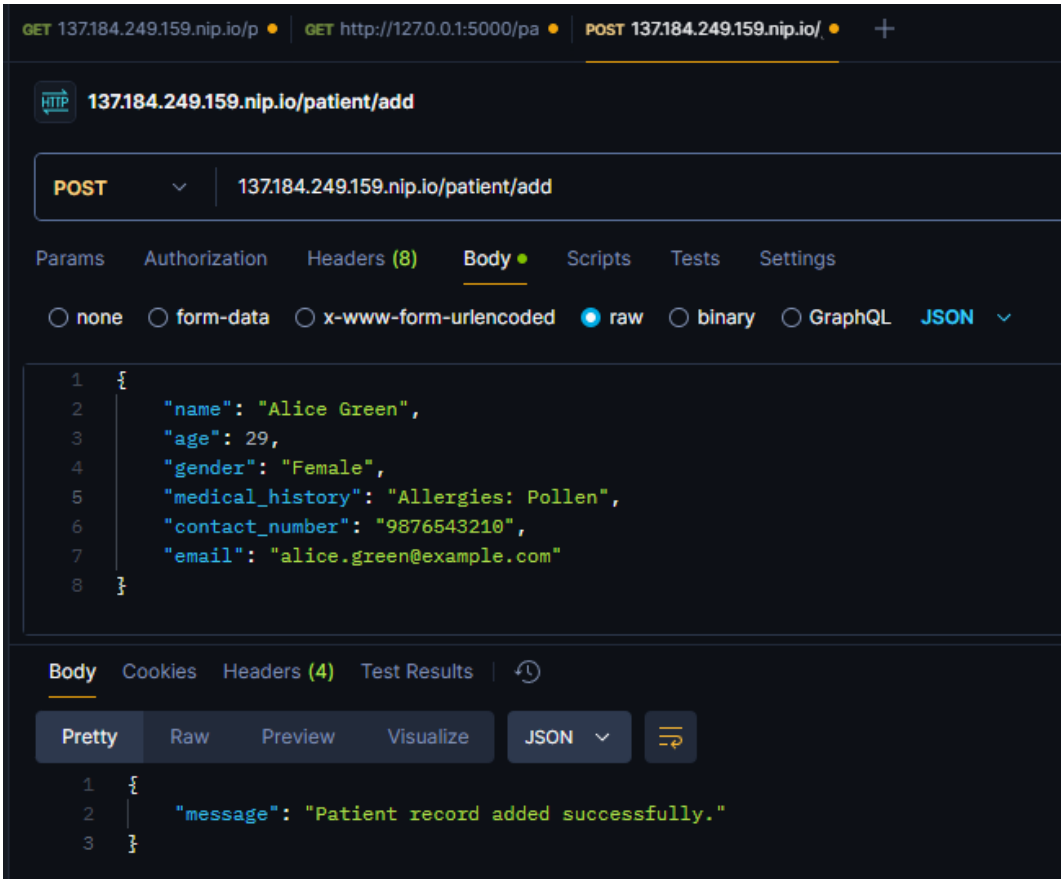

Figure 34. Add New Patient

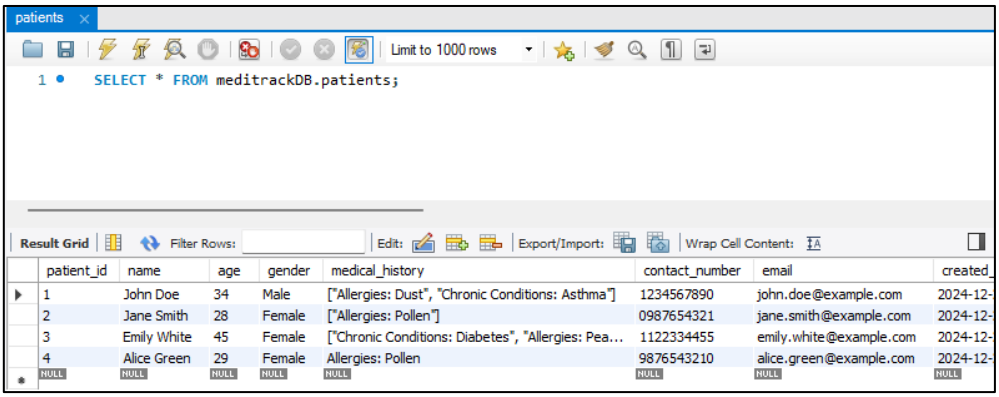

Figure 35. Verify in MySQL

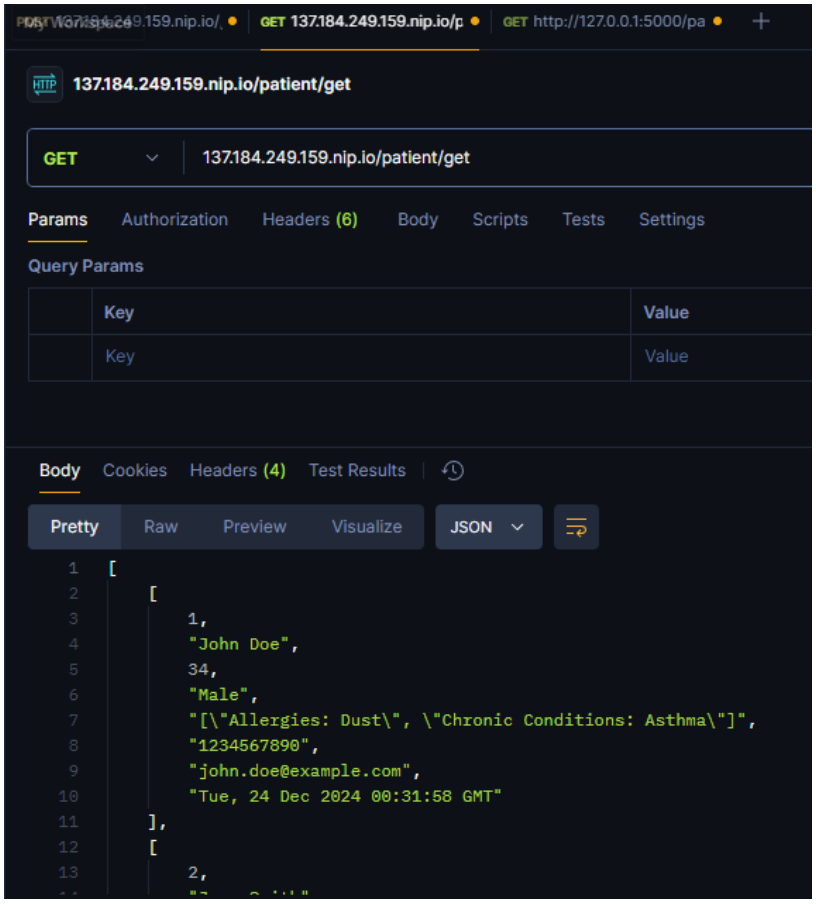

Figure 36. View all Patients

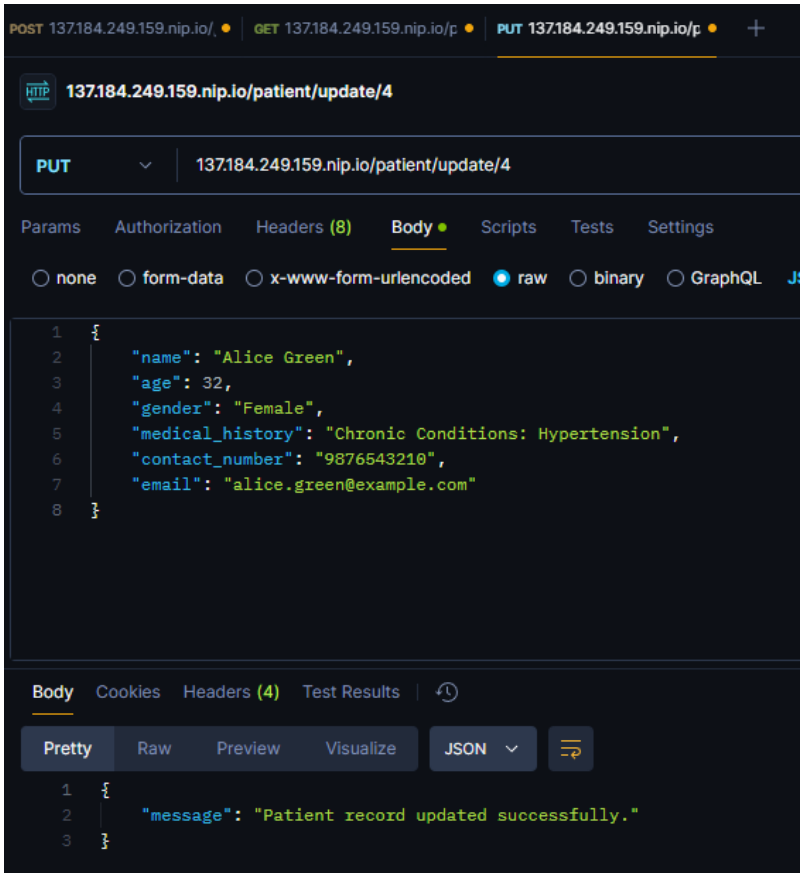

Figure 37. Update Patient Record

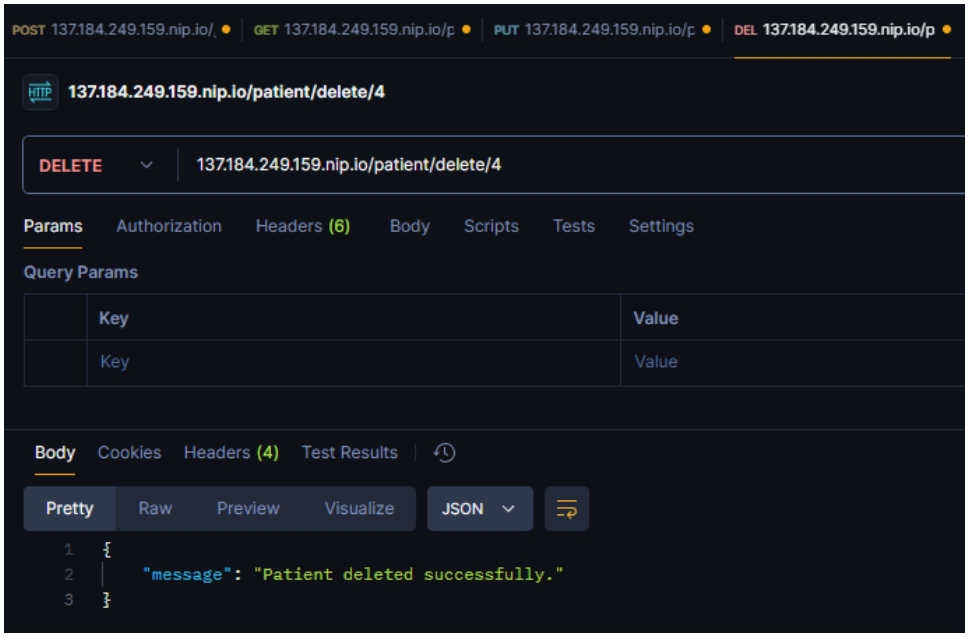

Figure 38. Delete Patient Using Patient ID

6. MediTrack System Deployment Runbook

6.1. Overview

This runbook provides step-by-step guidance for deploying and testing the MediTrack platform using a CI/CD pipeline on DigitalOcean. It covers setting up the environment, triggering the pipeline, deploying updates to **Blue and Green** environments, testing deployments, switching traffic, and handling failures with a robust fallback mechanism to ensure seamless and reliable application operations.

Runbook name	**MediTrack System Deployment Runbook**
Runbook description	Runbook to demonstrate the deployment and testing of the MediTrack system, which includes the following components: 1. **Patient Microservice**: Handles patient-related operations, including creating, retrieving, and updating patient details. This service is crucial for managing patient data securely and efficiently. 2. **Appointment Microservice**: Manages scheduling and tracking of appointments between patients and healthcare providers. This service ensures smooth appointment workflows. 3. **Notification Microservice**: Sends notifications (email/SMS) to patients and healthcare staff, ensuring timely communication. 4. **Aggregator Microservice**: Acts as a single-entry point for all microservices, integrating them into a cohesive system. It handles API requests and forwards them to the relevant microservices.
Owner	Raeesul Islam Zulfikar
Version	v1.0
Version date	25-12-2024
On this page	- Prerequisites - Deployment steps - Testing procedures - Troubleshooting

6.2. Support Contact

Expertise Level	Team	Contact
Developer and Owner	Raeesul Islam Zulfikar	Raeesul.20232953@iit.ac.lk

6.3. Process

Step	Task	Command/Action
Setting Up the Environment	**Verify Prerequisites** - Ensure Kubernetes cluster, DigitalOcean Container Registry, and GitHub CI/CD are set up.	Confirm infrastructure and access configurations.
	Configure Access - Authenticate with DigitalOcean and configure kubectl.	`doctl auth init` `doctl kubernetes cluster kubeconfig save <cluster-name>`

Trigger the CI/CD Pipeline	**Push Code Changes** - Push updates to the main branch of GitHub.	`git add .`
		`git commit -m "Update application code"`
		`git push origin main`

	Monitor Workflow - Verify CI/CD workflow trigger in GitHub.	Open GitHub repository and navigate to **Actions** tab.
Blue Deployment	**Switch Traffic** - Route traffic to Blue Environment via Blue Ingress Controller.	`kubectl describe ingress blue-ingress -n meditrack`
Build and Push Docker Images	**Build Docker Images** - CI/CD pipeline builds Docker images.	Check CI/CD logs in GitHub Actions for build status.
	Push to Registry - Docker images are pushed to DigitalOcean Container Registry.	Validate via DigitalOcean Container Registry dashboard.
Green Deployment	**Deploy Green** - Deploy updated images to Green Environment in Kubernetes.	`kubectl get deployments -n meditrack-green`
	Run Tests - Execute automated unit and integration tests in Green Environment.	`kubectl logs <pod-name> -n meditrack-green`

Switch to Green Environment	**Deploy to Blue** - Update Blue Environment with the stable deployment.	`kubectl get pods -n meditrack`
	Switch Traffic - Ingress Controller routes traffic to Green Environment.	Confirm ingress rules using kubectl describe ingress.
Post-Deployment Testing	**Testing APIs** - Use Postman to send API requests to the Green Environment.	`<green-ingress-ip>:<port>/api/<endpoint>`
	Validate Results - Check logs for errors and ensure proper responses.	`kubectl logs <pod-name> -n meditrack-green`
Fallback Mechanism	**Handle Failures** - Halt deployment and notify developers if Green Environment fails.	CI/CD pipeline logs indicate failure and notification to developers.
	Rollback Changes - Roll back Blue Environment if required.	`kubectl rollout undo deployment <deployment-name> -n meditrack`
	Retry Deployment - Fix issues, push updates, and retrigger CI/CD.	Follow **Step 2** to restart the process.

7. AWS Quick Sight Dashboard

Initially connect to AWS Redshift

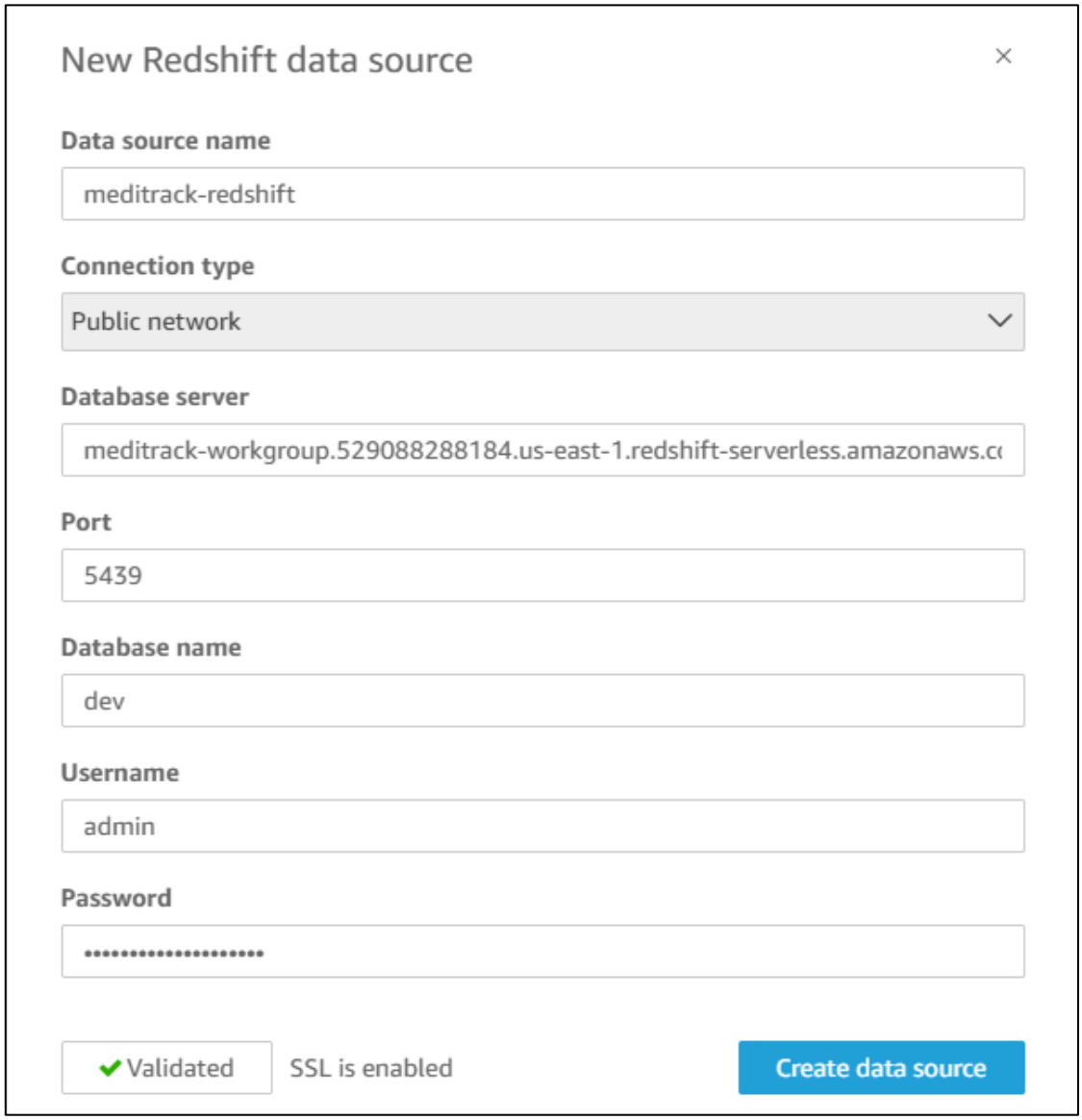

Figure 39. Connect Redshift to Quick Sight

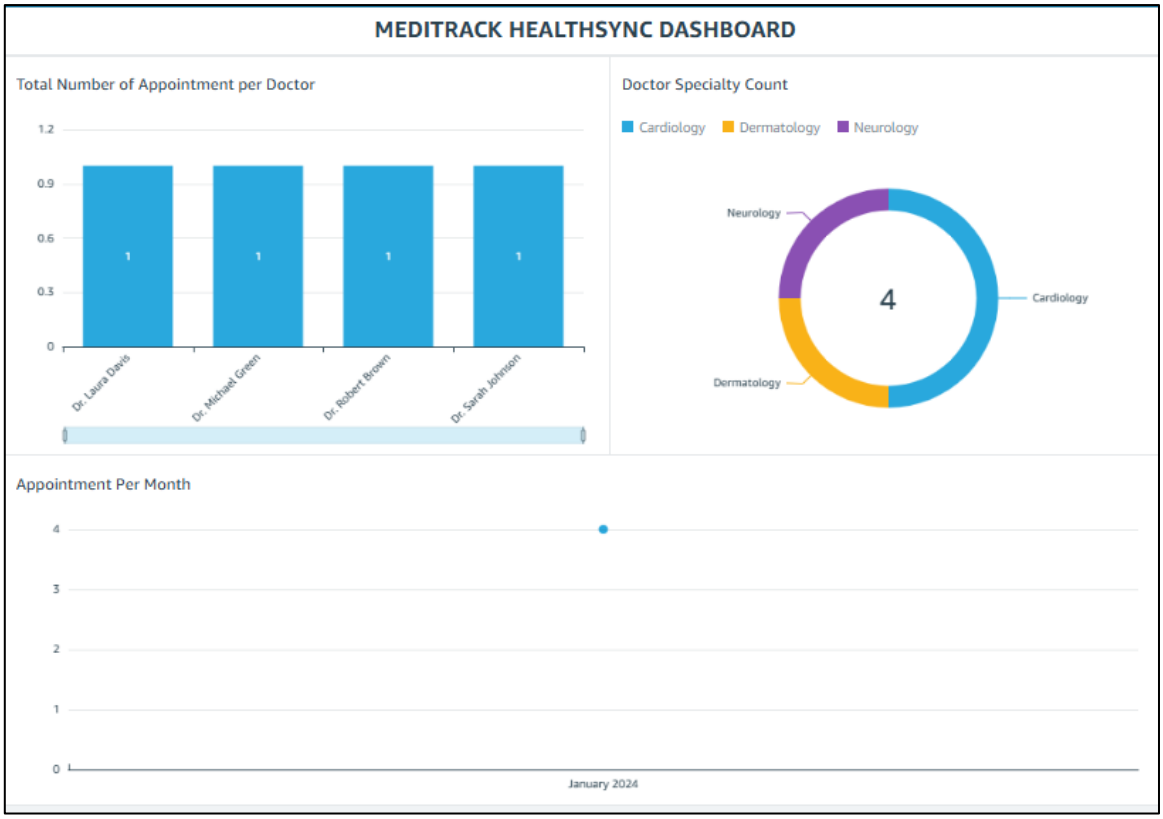

Figure 40. Meditrack Quick Sight Dashboard