CMM707 Cloud Computing

Coursework Report

MSc in Big Data Analytics

Submitted By
A.S.Fouzul Hassan
RGU No : 2410544 / IIT No : 20233214

Contents

1. System Architecture Diagram 3

1.1. Internal communication flow 4

2. Security & Ethical Challenges 5

3. CI/CD Process 6

4. Implementation 6

4.1. Microservices Implementation 6

4.2. Aggregator Service Implementation 10

4.3. CI/CD Process Implementation 13

4.4. CI/CD Deployment 15

5. Testing of the Deployed App 18

6. Observability 22

7. MoonInsurance System Deployment Runbook 24

7.1. Overview 24

7.2. Support Contacts 24

7.3. Process 25

8. Dashboard – MoonInsurance System 27

9. Metrics and High-Performance Analytics Support 28

1. System Architecture Diagram

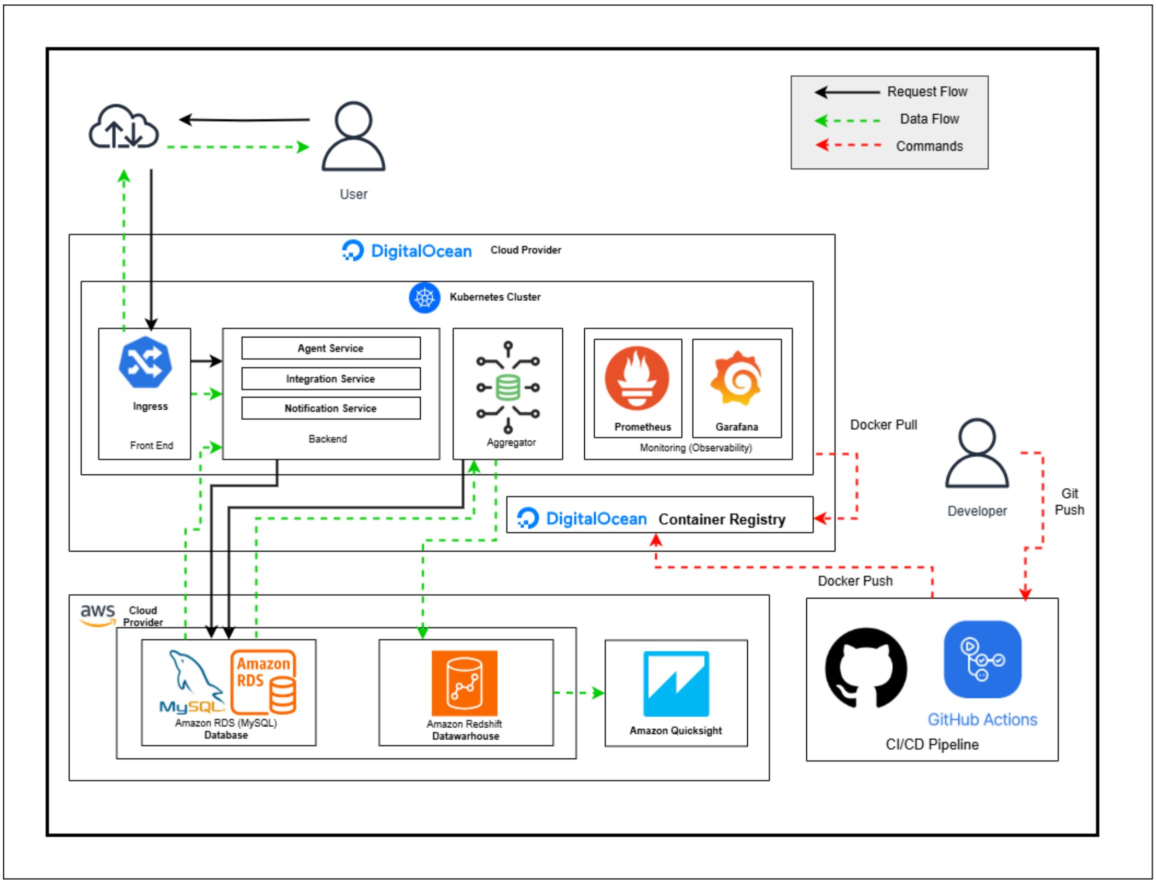


Figure 1- Solution Architecture Diagram for the System

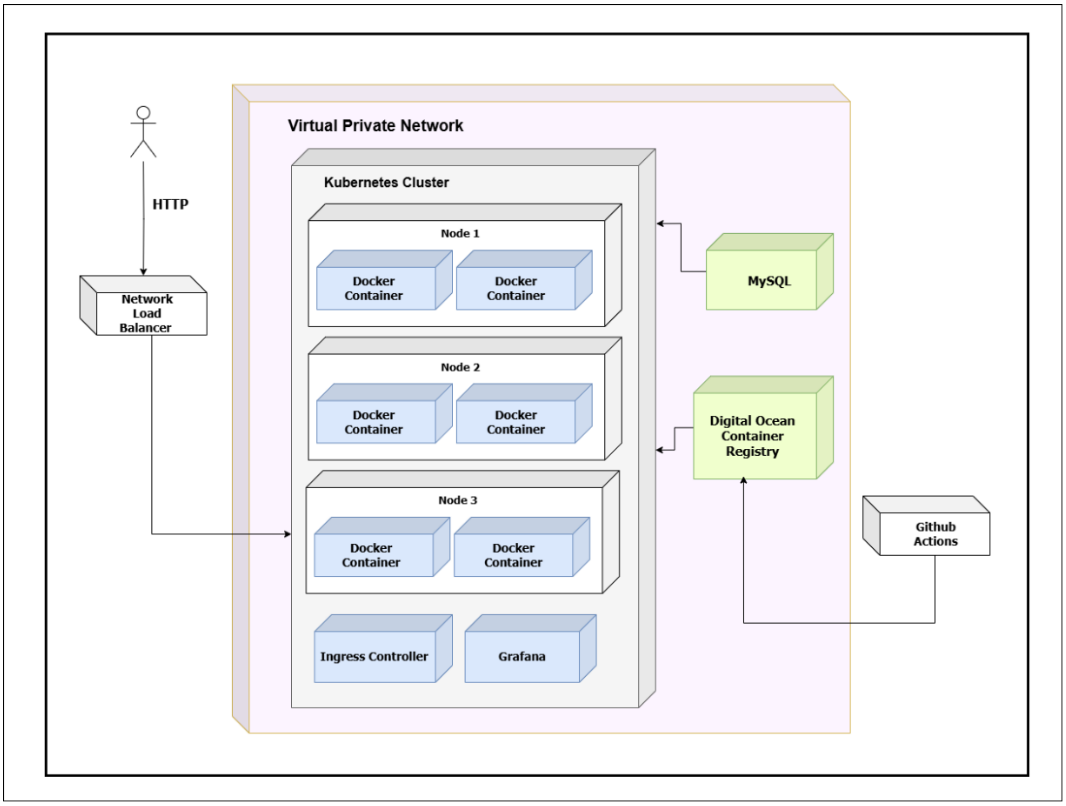


Figure 2 - Deployment Diagram

MoonInsurance is architected for scalability, security, fault-tolerance, and cost-efficiency. Docker images for every microservice are stored in the DigitalOcean Container Registry, while workloads are orchestrated by a managed DigitalOcean Kubernetes (DOKS) cluster.

- **Elastic scaling & HA** – Horizontal Pod Autoscalers, self-healing ReplicaSets and a regional Network Load Balancer keep services highly-available, while automatic node-pool scaling absorbs traffic spikes with zero-downtime Blue-Green releases.
- **Network & runtime isolation** – All service-to-service traffic is confined to a private VPC; an NGINX Ingress Controller terminates TLS and exposes only public APIs. Role-Based Access Control (RBAC), Kubernetes Secrets and DigitalOcean VPC firewalls gate access to runtime and data planes.
- **Observability** – Prometheus and Grafana (deployed via the Grafana Helm chart) stream metrics, logs and traces, raising alerts long before customers feel an impact.
- **Cost control** – DOKS pay-as-you-go pricing, spot-node pools for non-production workloads, and Open-Source monitoring tooling keep OpEx low.
- **Automated delivery** – A GitHub Actions pipeline builds, scans, and signs container images, pushes them to DOCR, then deploys them to a temporary *Green* namespace. Integration tests run in-cluster; on success traffic is flipped from *Blue* to *Green*. Rollback is instantaneous by re-pointing the Ingress.

1.1 Internal communication flow

1. **External requests**
   User → Cloud Load Balancer → NGINX Ingress → microservice Pods (Agent, Integration, Notification).
2. **Service discovery & east-west traffic**
   Pods talk to each other through ClusterIP services using Kubernetes DNS (agent-service.default.svc.cluster.local). mTLS can be toggled on with Linkerd if the compliance model tightens.
3. **Data layer**
   - **MySQL RDS** on AWS stores transactional data for agents and sales.
   - A scheduled **Aggregator CronJob** reads this data, computes KPIs (team sales, branch performance, product target attainment) and loads the results into **AWS Redshift Serverless**.
4. **Analytics & dashboards**
   Redshift is the single source of truth for analytics; **AWS QuickSight** consumes the aggregated tables to surface real-time dashboards for executives.
5. **CI/CD feedback loop**
   After every push, GitHub Actions → DOCR → DOKS. Deployment and test logs are shipped back to Grafana Loki, closing the observability loop.

This layered, modular communication pattern guarantees secure request routing, efficient data exchange, and hands-off operations—keeping MoonInsurance reliable as its user base and data volumes grow.

2. Security & Ethical Challenges

The **MoonInsurance** platform faces critical **security and ethical challenges** that must be addressed to maintain user trust, system integrity, and responsible data governance in the insurance domain. Since the platform processes sensitive information including policyholder identities, sales performance records, financial transactions, and team-level metrics, it becomes a potential target for cyber threats such as data breaches, credential theft, and ransomware.

From a **security** standpoint, the distributed microservices architecture—deployed via Kubernetes—introduces several vulnerabilities, especially during inter-service communication, API exposure, and data storage operations. Unauthorized access or data leakage during transmission could compromise customer confidentiality. To mitigate such risks, **end-to-end encryption** of data (both in-transit and at-rest), **API gateway rate limiting**, and **multi-factor authentication (MFA)** are enforced across all services. The use of **Role-Based Access Control (RBAC)** ensures agents, administrators, and developers can only access relevant information based on their roles.

Insider threats also pose a significant concern. Malicious or negligent access to sensitive sales or bonus data could disrupt fair agent compensation. This is mitigated through **auditing logs**, **user access reviews**, and **automated anomaly detection** integrated into the observability pipeline. All database connections are secured using environment-specific secrets stored securely through Kubernetes secrets management and GitHub Actions OIDC tokens for deployment.

On the **ethical** side, the MoonInsurance platform must handle **data ownership, fairness, and transparency** with care. Sales performance and aggregated metrics are used to make decisions about agent rankings, bonuses, and team performance—making it vital to ensure **data is not manipulated** or biased. The Aggregator microservice anonymizes data before analysis, and dashboards are permissioned in QuickSight to restrict visibility by role and region.

Furthermore, **algorithmic fairness** must be considered if predictive analytics or scoring systems are introduced. For instance, if the platform evolves to recommend training for underperforming agents based on AI predictions, it must ensure the model does not discriminate based on branch, gender, or historical bias. Similarly, insights derived from company-wide metrics should not be used for **commercial exploitation without consent** from agents or stakeholders.

The platform must remain transparent in how data is collected, processed, and used. Ethical safeguards such as **explicit consent flags**, **clear privacy notices**, and **data minimization** should be enforced. These practices must align with regulatory frameworks such as **Sri Lanka's Personal Data Protection Act (PDPA)** and other international data standards if the platform scales globally.

As a conclusion, MoonInsurance must adopt a **proactive and ethical-first approach** to both security and data governance. This includes:

- Implementing **robust security protocols**
- Maintaining **transparent ethical practices**
- Conducting **regular penetration tests and compliance audits**
- Enabling **real-time monitoring and alerts**
- Educating users on **privacy and responsible data handling**

Such measures not only improve resilience and reliability but also ensure the platform upholds high standards of **trust, fairness, and accountability** in the insurance technology sector.

3. CI/CD Process

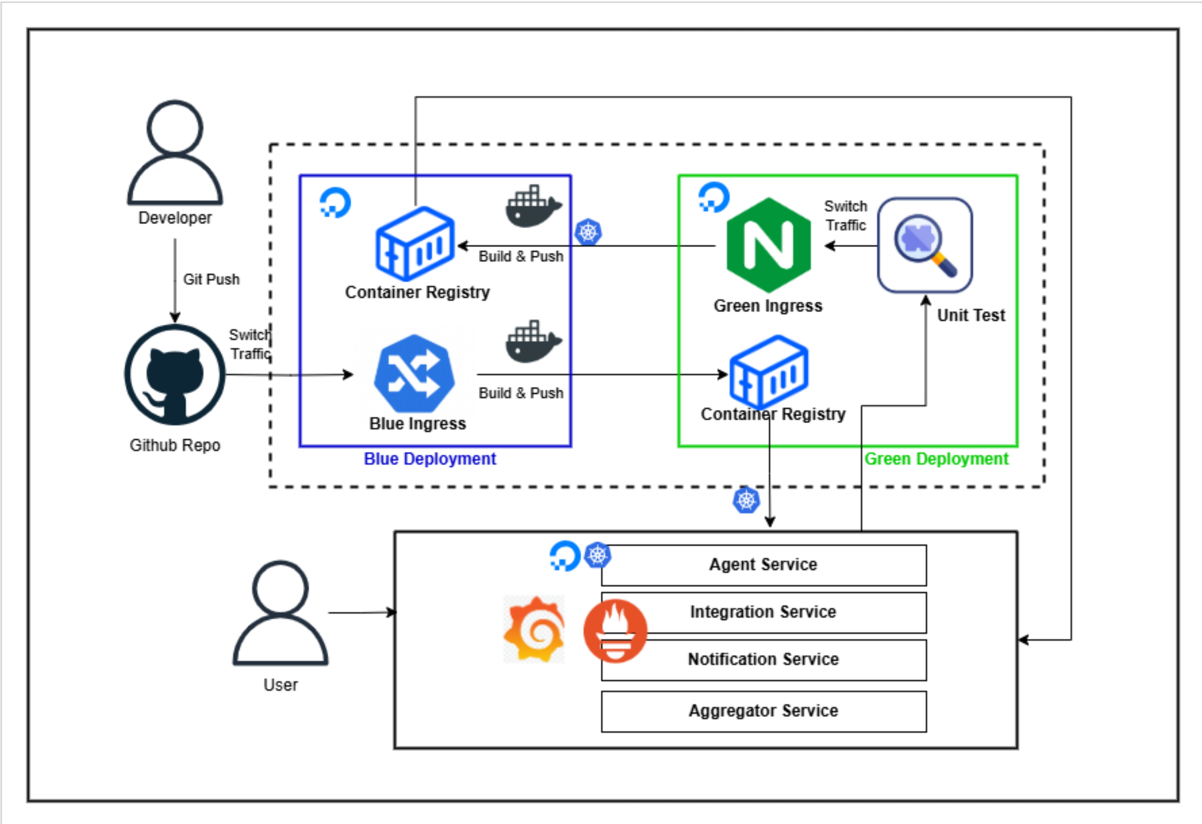


Figure 3 - Diagram for CI/CD Process

4. Implementation

4.1. Microservices Implementation

Initially a MySQL Database is been created in Amazon RDS and connected remotely to the MySQL Workbench.

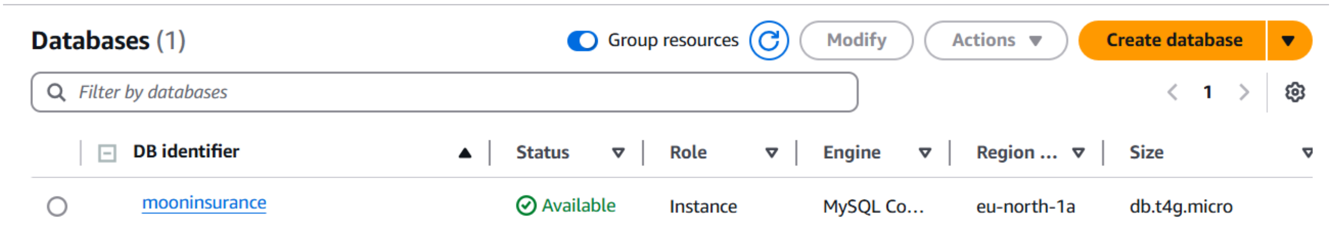


Figure 4 - Amazon RDS (MySQL)

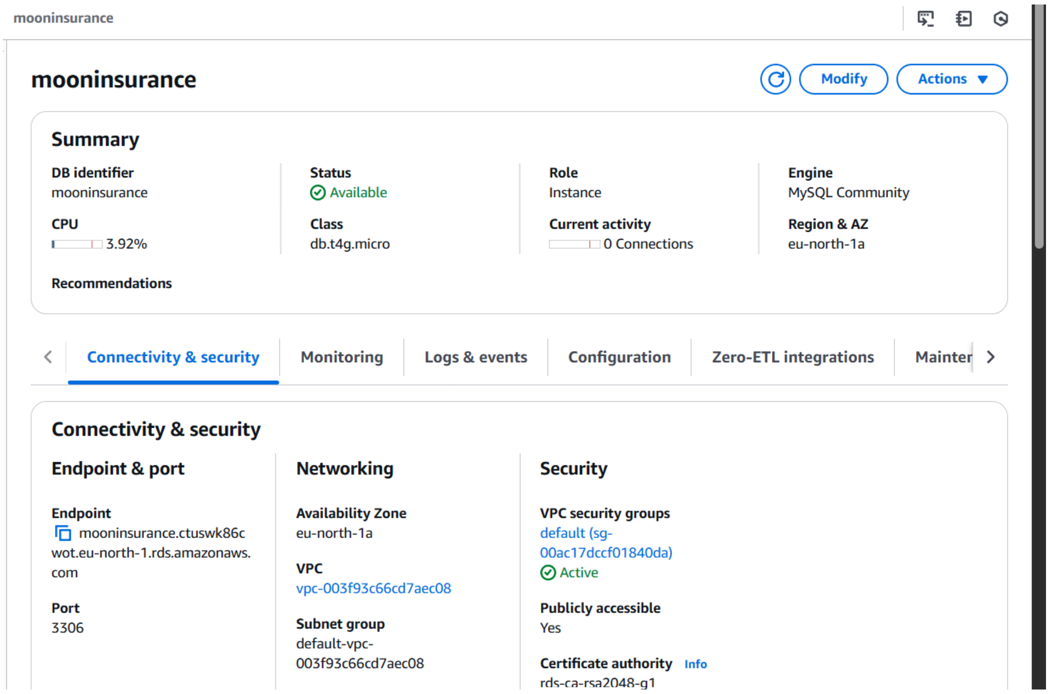


Figure 5 - Amazon RDS (MySQL) Details

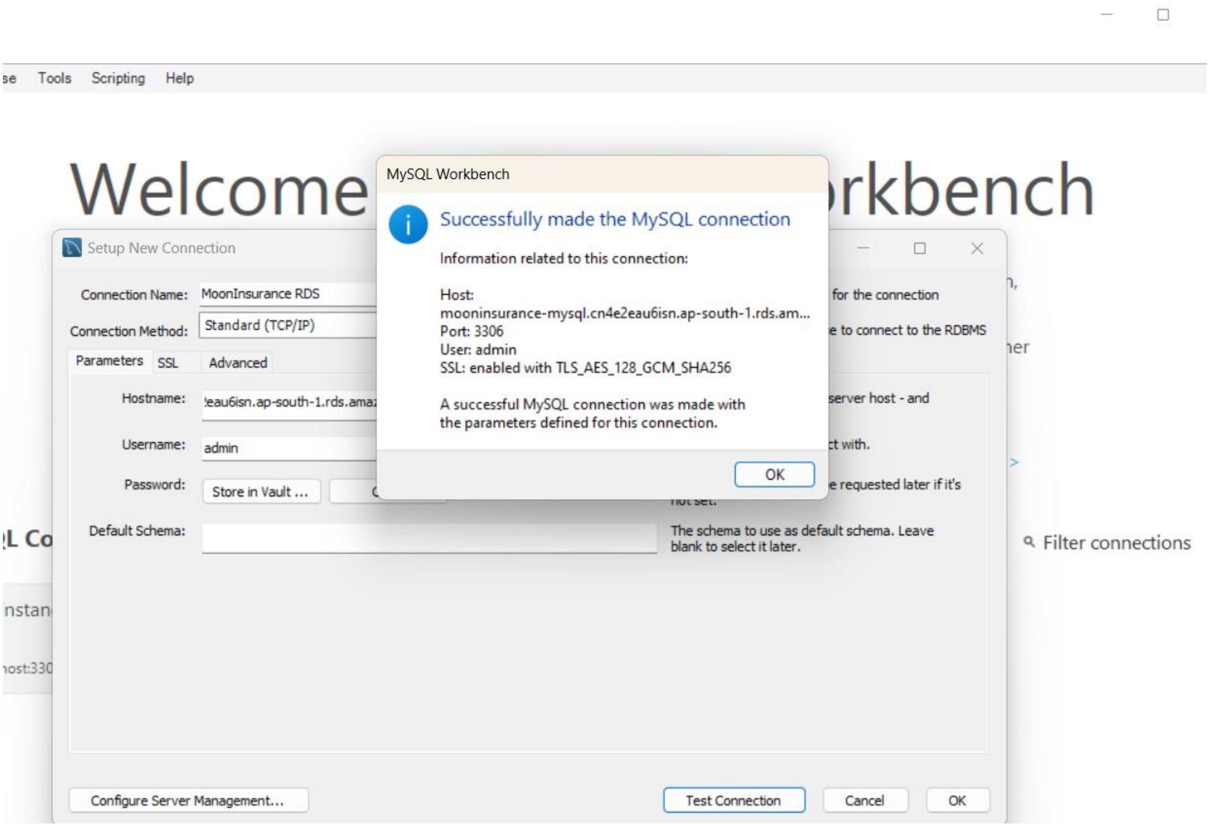
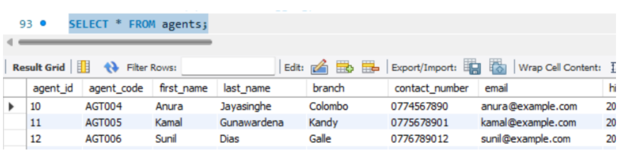
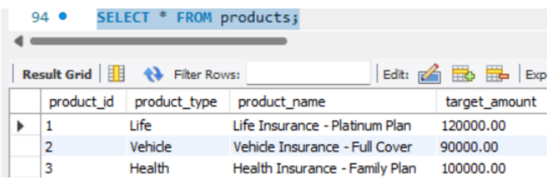


Figure 6 - Connecting AWS RDS to MySQL Workbench

Database is created and also few dummy data is inserted into the database.





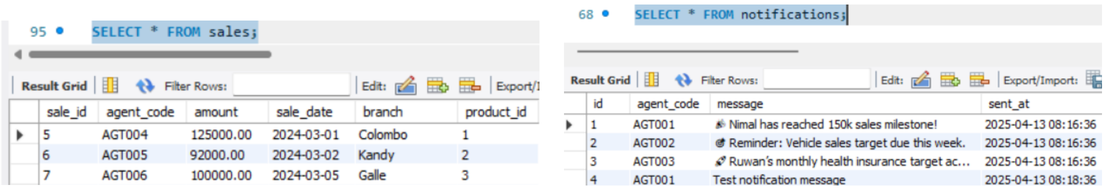


Figure 7 - Few Screenshots of Implemented Tables

Microservice code API is developed with Python Flask.

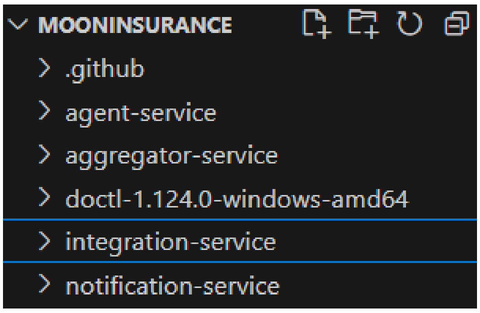


Figure 8 - File Structure of the solution

Figure 9 - Agent Services-Code Implementation

integration-service > integration_app.py > ...

```python

@app.route('/integration')
def index():
    return render_template('index.html')

# CREATE: Add a new sale
@app.route('/integration/add', methods=['POST'])
def add_sale():
    data = request.json
    print("Received Agent Data:", data)
    try:
        connection = mysql.connector.connect(
            host=MYSQL_HOST, user=MYSQL_USER,
            password=MYSQL_PWD, database=MYSQL_DB, port=PORT
        )
        cursor = connection.cursor()

        query = """
        INSERT INTO sales (agent_code, product_id, amount, sale_date, branch)
        VALUES (%s, %s, %s, %s, %s)
        """
        cursor.execute(query, (
            data.get('agent_code'),
            data.get('product_id'),
            data.get('amount'),
            data.get('sale_date'),
            data.get('branch')
        ))
        connection.commit()
        return jsonify({"message": "Sale record added successfully."}), 201

    except mysql.connector.Error as err:
```

Figure 10 - Integration Service Code-Implementation

notification-service > notification_app.py > send_notification

```python
@app.route('/notification')
def index():
    return render_template('index.html')

# Send a notification
@app.route('/notification/send', methods=['POST'])
def send_notification():
    data = request.json
    agent_code = data.get("agent_code")
    message = data.get("message")

    try:
        connection = mysql.connector.connect(
            host=MYSQL_HOST,
            user=MYSQL_USER,
            password=MYSQL_PWD,
            database=MYSQL_DB,
            port=PORT
        )
        cursor = connection.cursor()
        query = "INSERT INTO notifications (agent_code, message) VALUES (%s, %s)"
        cursor.execute(query, (agent_code, message))
        connection.commit()
        return jsonify({"message": f"Notification sent to agent {agent_code}."}), 201

    except mysql.connector.Error as err:
        print(f"Error: {err}")
        return jsonify({"message": "Failed to send notification."}), 500

# Get all notifications
@app.route('/notification/get', methods=['GET'])
def get_notifications():
```

Figure 11 - Notification Service Implmentation

4.2. Aggregator Service Implementation

AWS Redshift Namespace and workgroups are created.

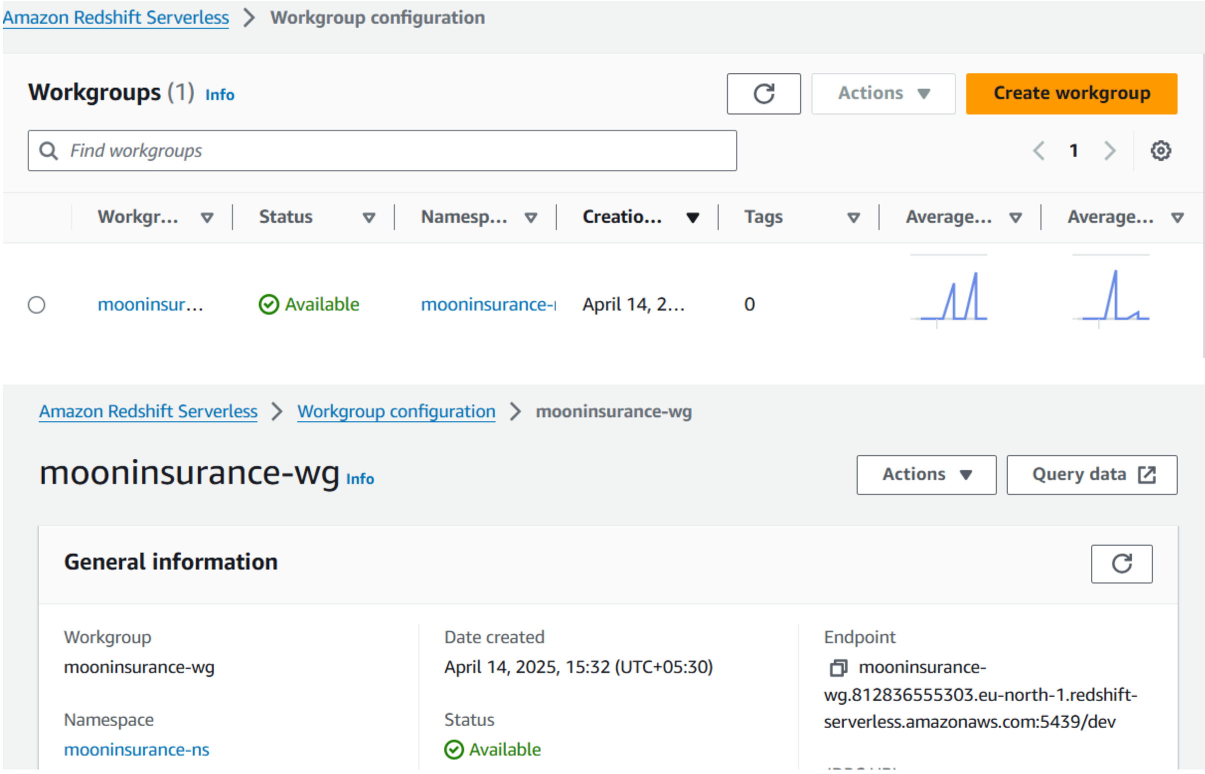
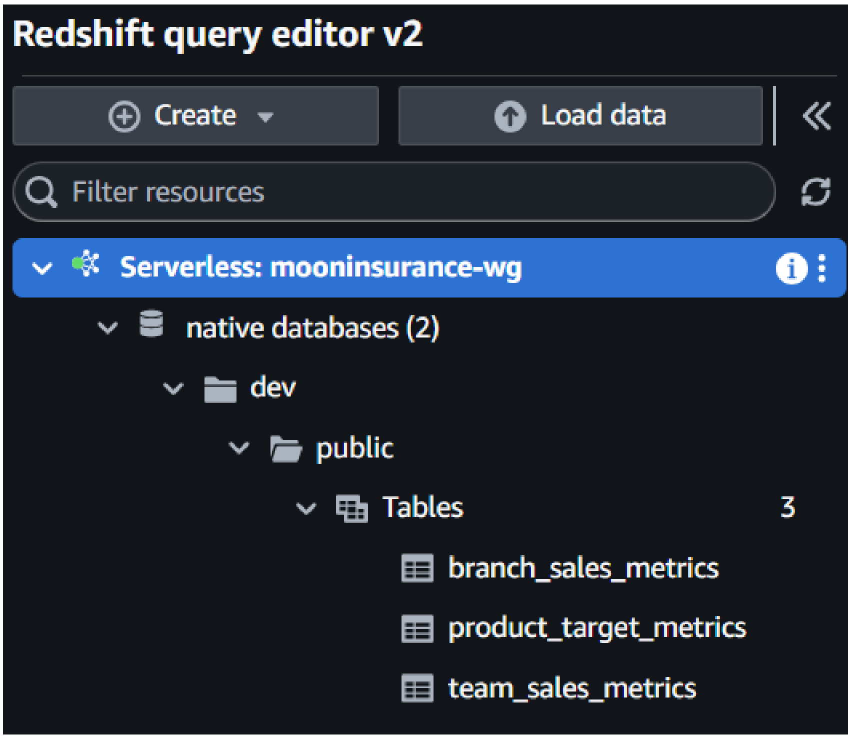


Figure 12 - AWS-Redshift

A Data Warehouse is created in Redshift for Aggregation Service.



An API is developed for aggregation, where the codes writes the data to the Datawarehouse.

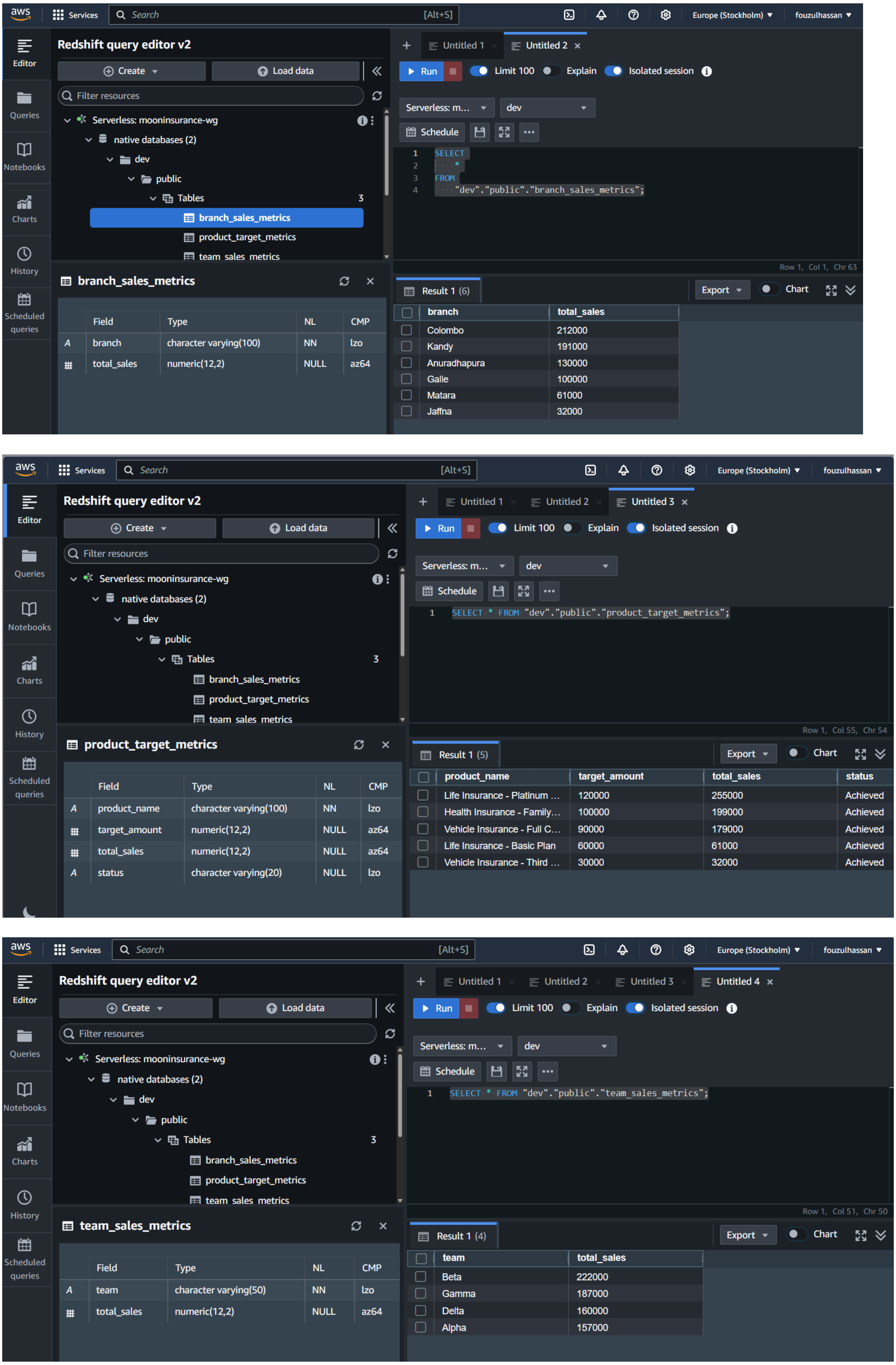


Figure 13 - Redshift Query Editor (Datawarehouse)

```python
def fetch_dataframe(query):
    try:
        conn = mysql.connector.connect(**MYSQL_CONFIG)
        df = pd.read_sql(query, conn)
        conn.close()
        return df
    except mysql.connector.Error as err:
        print(f"[MySQL ERROR] {err}")
        return pd.DataFrame()

def get_team_sales():
    return fetch_dataframe("""
        SELECT a.team, SUM(s.amount) AS total_sales
        FROM agents a
        JOIN sales s ON a.agent_code = s.agent_code
        GROUP BY a.team
        ORDER BY total_sales DESC
    """)

def get_product_target_achievement():
    return fetch_dataframe("""
        SELECT p.product_name, p.target_amount,
               COALESCE(SUM(s.amount), 0) AS total_sales,
               CASE
                   WHEN COALESCE(SUM(s.amount), 0) >= p.target_amount
                   THEN 'Achieved' ELSE 'Not Achieved'
               END AS status
        FROM products p
        LEFT JOIN sales s ON p.product_id = s.product_id
        GROUP BY p.product_id
```

Figure 14 - Fetch Aggregations

```python
def connect_redshift():
    return psycopg2.connect(**REDSHIFT_CONFIG)

def upsert_team_sales(df):
    conn = connect_redshift()
    cursor = conn.cursor()
    for _, row in df.iterrows():
        cursor.execute("""
            DELETE FROM team_sales_metrics WHERE team = %s;
        """, (row['team'],))
        cursor.execute("""
            INSERT INTO team_sales_metrics (team, total_sales)
            VALUES (%s, %s);
        """, (row['team'], row['total_sales']))
    conn.commit()
    cursor.close()
    conn.close()
    print("✅ Upserted team_sales_metrics")

def upsert_product_target_metrics(df):
    conn = connect_redshift()
    cursor = conn.cursor()
    for _, row in df.iterrows():
        cursor.execute("""
            DELETE FROM product_target_metrics WHERE product_name = %s;
        """, (row['product_name'],))
        cursor.execute("""
            INSERT INTO product_target_metrics (product_name, target_amount, total_sales,
            VALUES (%s, %s, %s, %s);
        """, (row['product_name'], row['target_amount'], row['total_sales'], row['status']
    conn.commit()
```

```python
def upsert_branch_sales(df):
    conn = connect_redshift()
    cursor = conn.cursor()
    for _, row in df.iterrows():
        cursor.execute("""
            DELETE FROM branch_sales_metrics WHERE branch = %s;
        """, (row['branch'],))
        cursor.execute("""
            INSERT INTO branch_sales_metrics (branch, total_sales)
            VALUES (%s, %s);
        """, (row['branch'], row['total_sales']))
    conn.commit()
    cursor.close()
    conn.close()
    print("✅ Upserted branch_sales_metrics")
```

Figure 15 - Insert into Redshift

4.3. CI/CD Process Implementation

```
# Set working directory
WORKDIR /app

# Copy the service files
COPY . .

RUN pip install --no-cache-dir -r requirements.txt

EXPOSE 5000

CMD ["gunicorn", "-b", "0.0.0.0:5000", "agent_app:app"]
```

Figure 16 - Docker File

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

    spec:
      containers:
      - name: agent-service
        image: registry.digitalocean.com/moonregistry/agent-service:blue-v2
        ports:
        - containerPort: 80
      imagePullSecrets:
      - name: do-secret

---
apiVersion: v1
kind: Service
metadata:
  name: agent-service-blue
spec:
```

Figure 17 - Blue Ingress Deployment (Yaml)

```yaml
io.k8s.api.core.v1.Service (v1@service.json) | io.k8s.api.apps.v1.Deployment (v1@deployment.json)
apiVersion: apps/v1
kind: Deployment
metadata:
  name: agent-service-green
spec:
  replicas: 1
  selector:
    matchLabels:
      app: agent-service
      version: green
  template:
    metadata:
      labels:
        app: agent-service
        version: green

    spec:
      containers:
      - name: agent-service
        image: registry.digitalocean.com/moonregistry/agent-service:green-v2
        ports:
        - containerPort: 80
      imagePullSecrets:
      - name: do-secret

---
apiVersion: v1
kind: Service
metadata:
  name: agent-service-green
spec:
```

Figure 18 - Green Ingress Deployment (Yaml)

Similarly, this is implemented for all the three services.

```yaml
io.k8s.api.batch.v1.CronJob (v1@cronjob.json)
apiVersion: batch/v1
kind: CronJob
metadata:
  name: aggregator-cronjob
spec:
  schedule: "30 18 * * *"  # Runs every day at 6:30 PM UTC (12 AM in Sri Lanka)
  jobTemplate:
    spec:
      template:
        spec:
          containers:
            - name: aggregator
              image: registry.digitalocean.com/meditrackcontainer/aggregator-service:latest
              imagePullPolicy: Always
          restartPolicy: Never
          imagePullSecrets:
            - name: do-secret
```

Figure 19 - Cron-job of AggregationService

```yaml
! ingress-blue.yaml > {} spec > [ ] rules > {} 0 > {} http > [ ] paths > {}
io.k8s.api.networking.v1.Ingress (v1@ingress.json)
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: microservices-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target:
spec:
  ingressClassName: nginx
  rules:
  - host: 209.38.124.165.nip.io
    http:
      paths:
      - path: /agent
        pathType: Exact
        backend:
          service:
            name: agent-service-blue
            port:
              number: 80
      - path: /agent/get
        pathType: Exact
        backend:
          service:
            name: agent-service-blue
            port:
              number: 80
      - path: /agent/add
        pathType: Exact
        backend:
          service:
            name: agent-service-blue
```

```yaml
! ingress-green.yaml > {} spec > [ ] rules > {} 0 > {} http > [ ] paths > {} 1
io.k8s.api.networking.v1.Ingress (v1@ingress.json)
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: microservices-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target:
spec:
  ingressClassName: nginx
  rules:
  - host: 209.38.124.165.nip.io
    http:
      paths:
      - path: /agent
        pathType: Exact
        backend:
          service:
            name: agent-service-green
            port:
              number: 80
      - path: /agent/get
        pathType: Exact
        backend:
          service:
            name: agent-service-green
            port:
              number: 80
      - path: /agent/add
        pathType: Exact
        backend:
          service:
            name: agent-service-green
```

Figure 20 - Green and Blue Ingress-YAML

4.4. CI/CD Deployment

CI (Build & Test):
Upon each code push to GitHub, the system automatically triggers GitHub Actions to build Docker images for all microservices (Agent, Integration, Notification, Aggregator) and execute automated tests to validate them.

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

CD (Deploy):
After a successful build and test phase, updated services are deployed to the DigitalOcean Kubernetes cluster.

- Deployment begins with the **Green** environment.
- Stability tests are conducted post-deployment.
- Traffic is switched from **Blue** to **Green** following the **Blue-Green deployment strategy**.
- The kubectl rollout restart command is used to restart the deployments and apply updates.
- The **Aggregator Service** is deployed as a **CronJob**, enabling periodic data aggregation and transfer to Redshift.

Actual Workflow (from YAML file)

1. change-to-blue : Build & push aggregator & Apply cronjob and ingress-blue

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

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

      - name: Log in to DigitalOcean
        run: echo "${{ secrets.DIGITALOCEAN_ACCESS_TOKEN }}" | docker login reg

      - name: Build and push Aggregator Service
        run: |
          docker build --no-cache -t registry.digitalocean.com/moonregistry/agg
          docker push registry.digitalocean.com/moonregistry/aggregator-service

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

      - name: Install kubectl
        uses: azure/setup-kubectl@v3
```

2. build-green: Build & push agent, integration, notification (green)

```yaml
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: Configure kubeconfig
      run: doctl kubernetes cluster kubeconfig save mooninsurance-cluster

    - name: Apply Green Deployments
      run: |
        kubectl apply -f ./agent-service/agent-green-deployment.yaml
        kubectl apply -f ./integration-service/integration-green-deployment.yaml
        kubectl apply -f ./notification-service/notification-green-deployment.yaml
```

3. deploy-green: Deploy all green services to K8s using kubectl

```yaml
test-green:
  needs: deploy-green
  runs-on: ubuntu-latest

  steps:
    - name: Wait for green deployments to stabilize
      run: sleep 30

    - name: Test Agent Service (Green)
      run: |
        curl -f http://209.38.124.165.nip.io/agent/get || exit 1

    - name: Test Integration Service (Green)
      run: |
        curl -f http://209.38.124.165.nip.io/integration/get || exit 1

    - name: Test Notification Service (Green)
      run: |
        curl -f http://209.38.124.165.nip.io/notification/get || exit 1
```

4. build-blue:P Prepare next version of blue (agent/integration/notification)

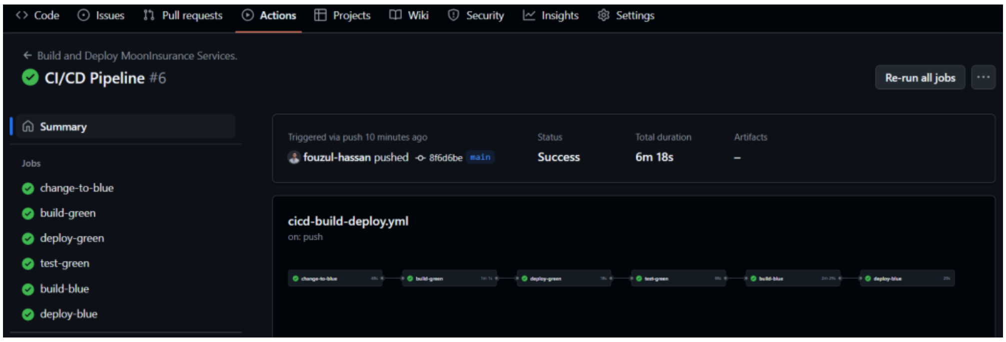
```yaml
deploy-blue:
  needs: build-blue
  runs-on: ubuntu-latest

  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: Configure kubeconfig
      run: doctl kubernetes cluster kubeconfig save mooninsurance-cluster

    - name: Apply Blue Deployments and Ingress
      run: |
        kubectl apply -f ./agent-service/agent-blue-deployment.yaml
        kubectl apply -f ./integration-service/integration-blue-deployment.yaml
        kubectl apply -f ./notification-service/notification-blue-deployment.yaml
        kubectl apply -f ingress-green.yaml
        kubectl rollout restart deployment agent-service-blue
        kubectl rollout restart deployment integration-service-blue
        kubectl rollout restart deployment notification-service-blue
```

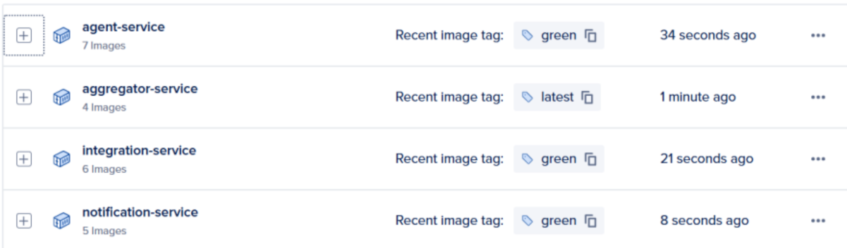
5. deploy-blue: Restart blue deployments &Switch ingress if needed

```yaml
deploy-blue:
  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: Configure kubeconfig
      run: doctl kubernetes cluster kubeconfig save mooninsurance-cluster
```

Blue-Green Deployment : These are two versions (Blue & Green) run in parallel which only one handles **live traffic**. The **inactive ones are been deployed**, test it, then switch

This ensures there is no any downtime, enables Automated builds and deployments, Safe rollback and Observability.






```
PS C:\MSc Files\Cloud Computing\CW\Solution\MoonInsurance> kubectl get pods
NAME                                          READY   STATUS    RESTARTS   AGE
agent-service-blue-5d848ffffb-ns65x           1/1     Running   0          5h34m
agent-service-green-56b47787cf-rw7q8          1/1     Running   0          5h36m
integration-service-blue-856ffc86c-sgz55      1/1     Running   0          5h34m
integration-service-green-7cd466d8c7-kh8qw    1/1     Running   0          5h36m
notification-service-blue-846699774f-cdcpj    1/1     Running   0          5h34m
notification-service-green-84cb8ffd6f-s9nhb   1/1     Running   0          5h36m
```

5. Testing of the Deployed App

Testing the deployed app using Postman.

1. Agent Service

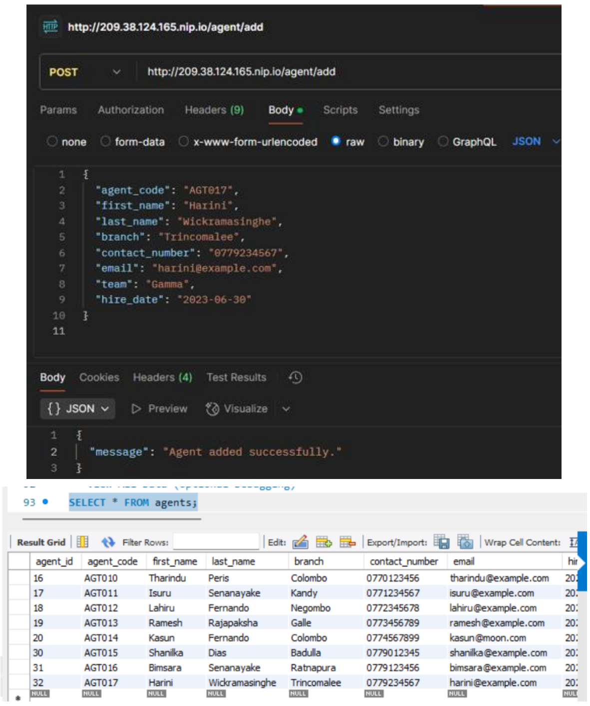


Figure 21 - Insert New-Agent and Verify

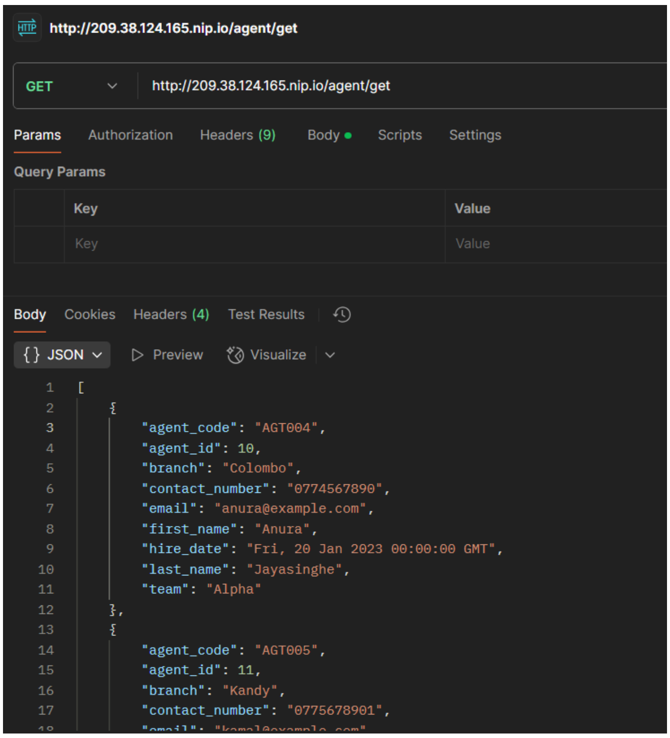


Figure 22-View-All Agents

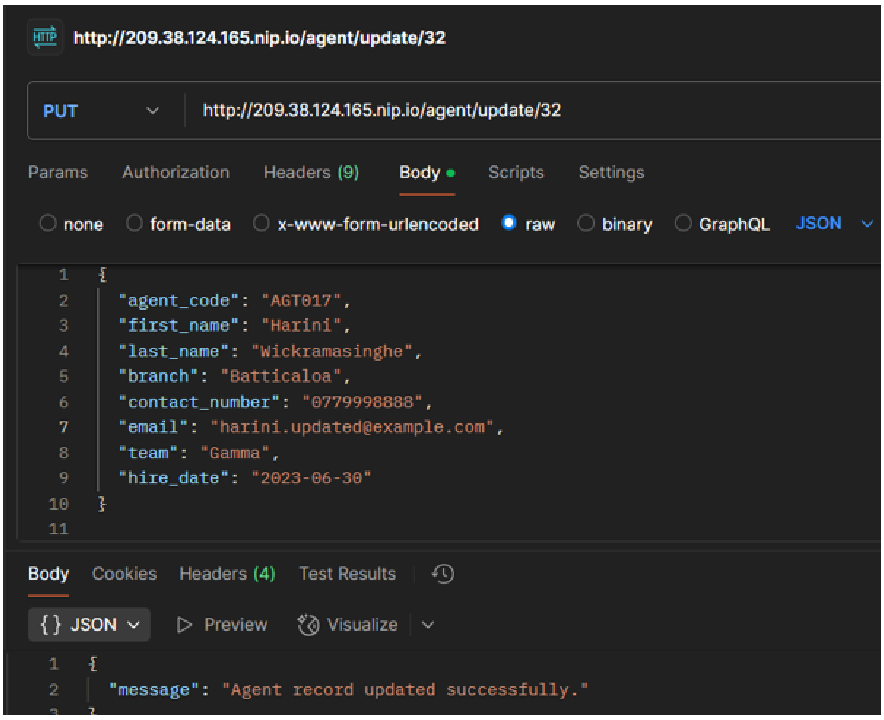


Figure 23 - Update-Agent

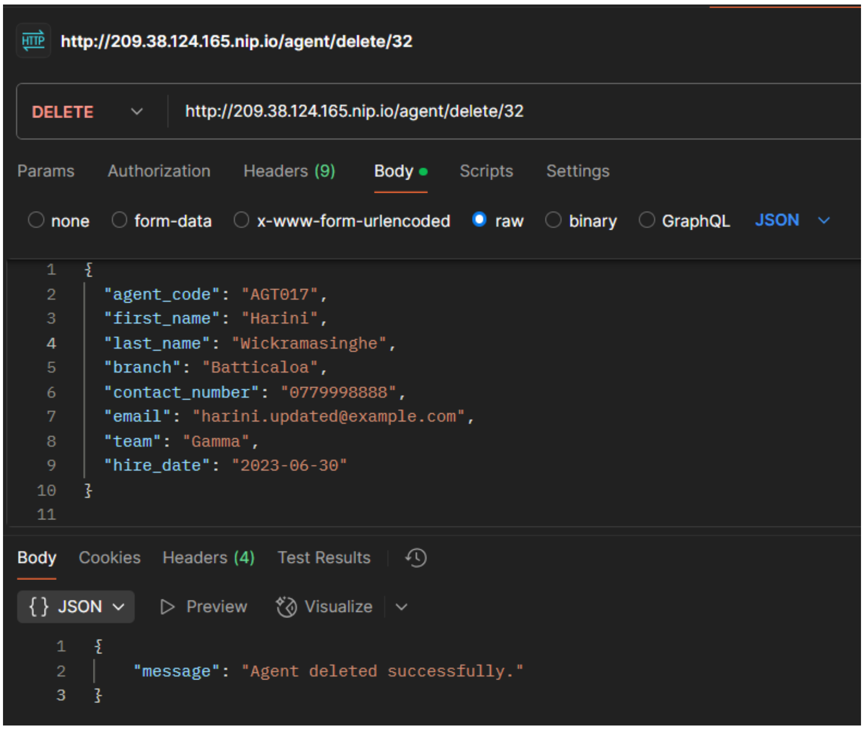


Figure 24 - Delete Agent

2. Integration Service

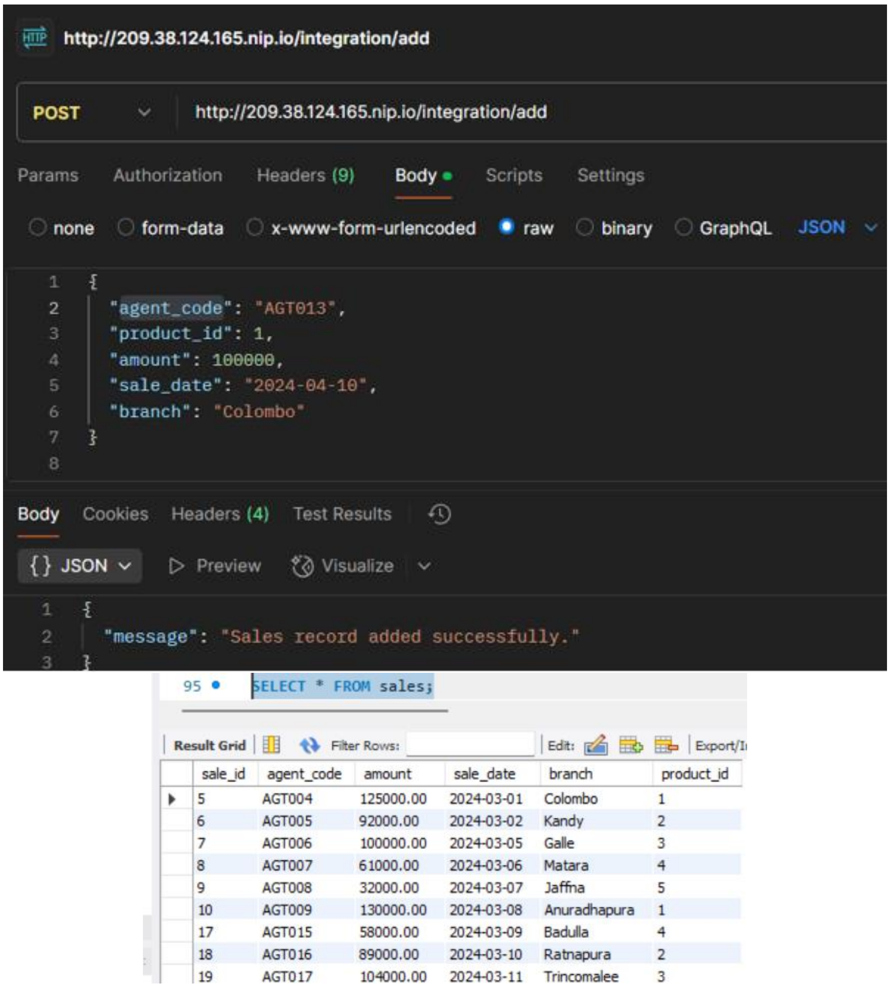


Figure 25 - Add-New Integration/Sales & Verify

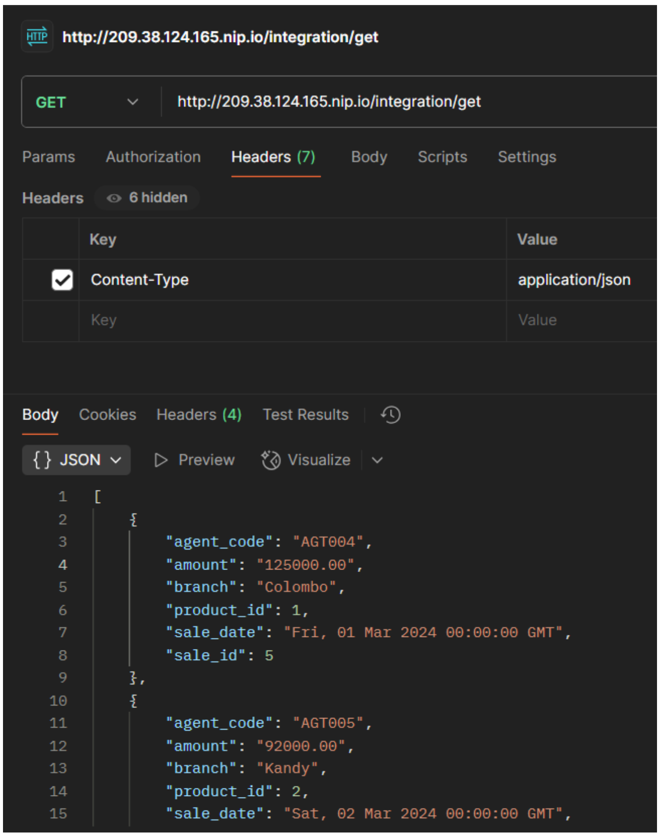


Figure 26 - View-all Integration/Sales

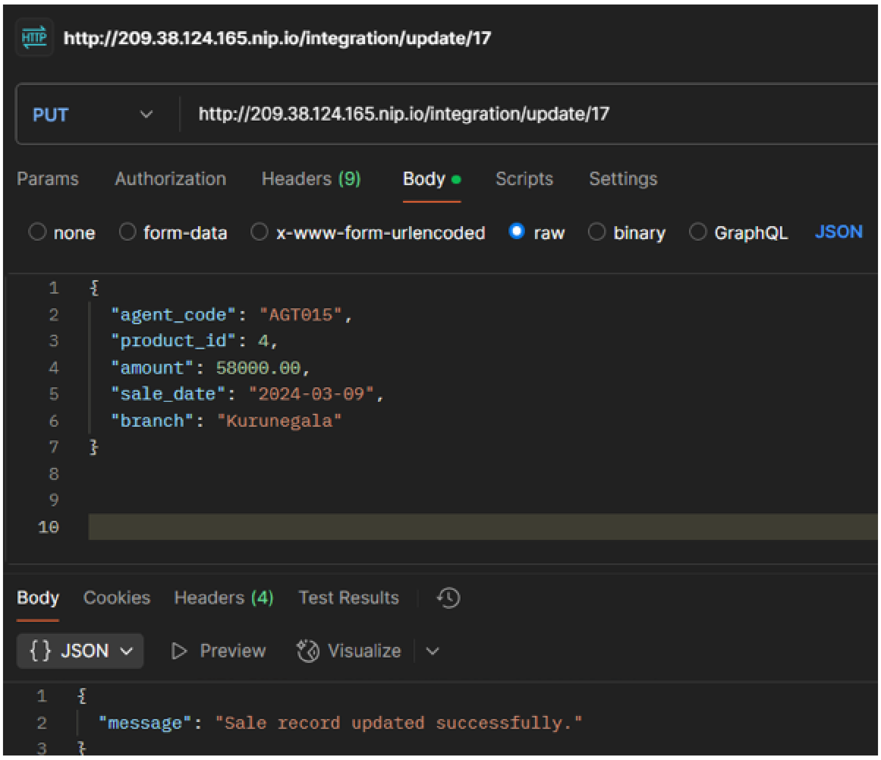


Figure 27 - Update-Integration/Sales

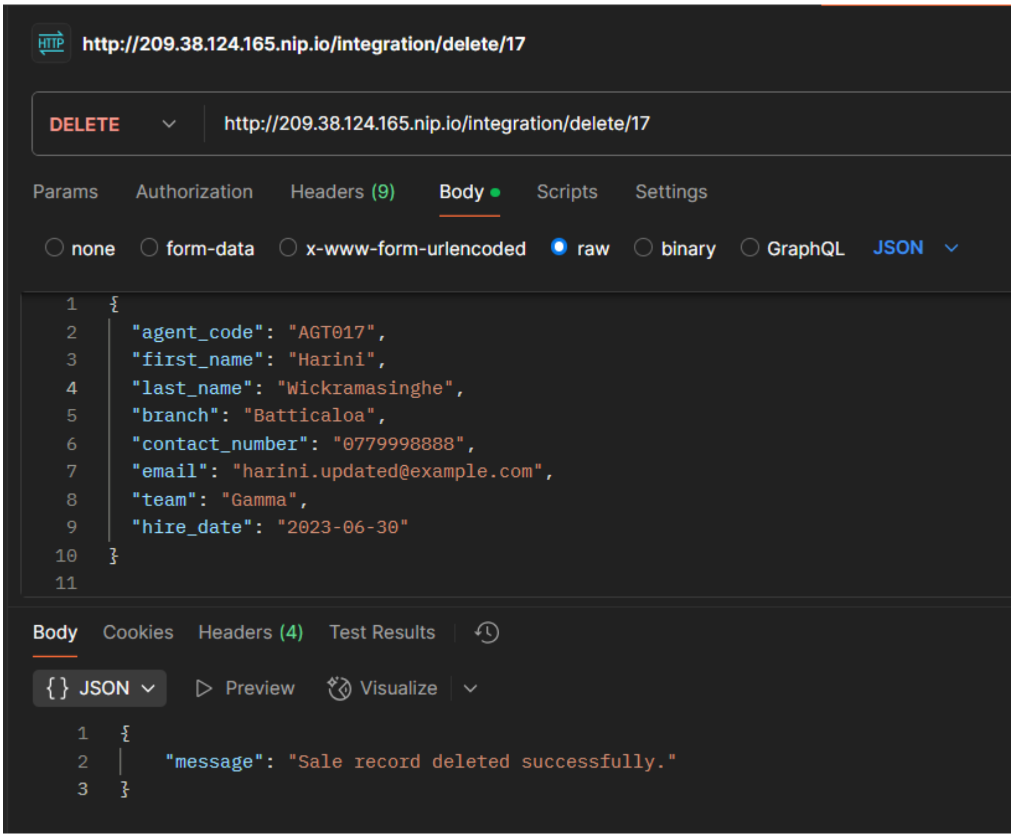
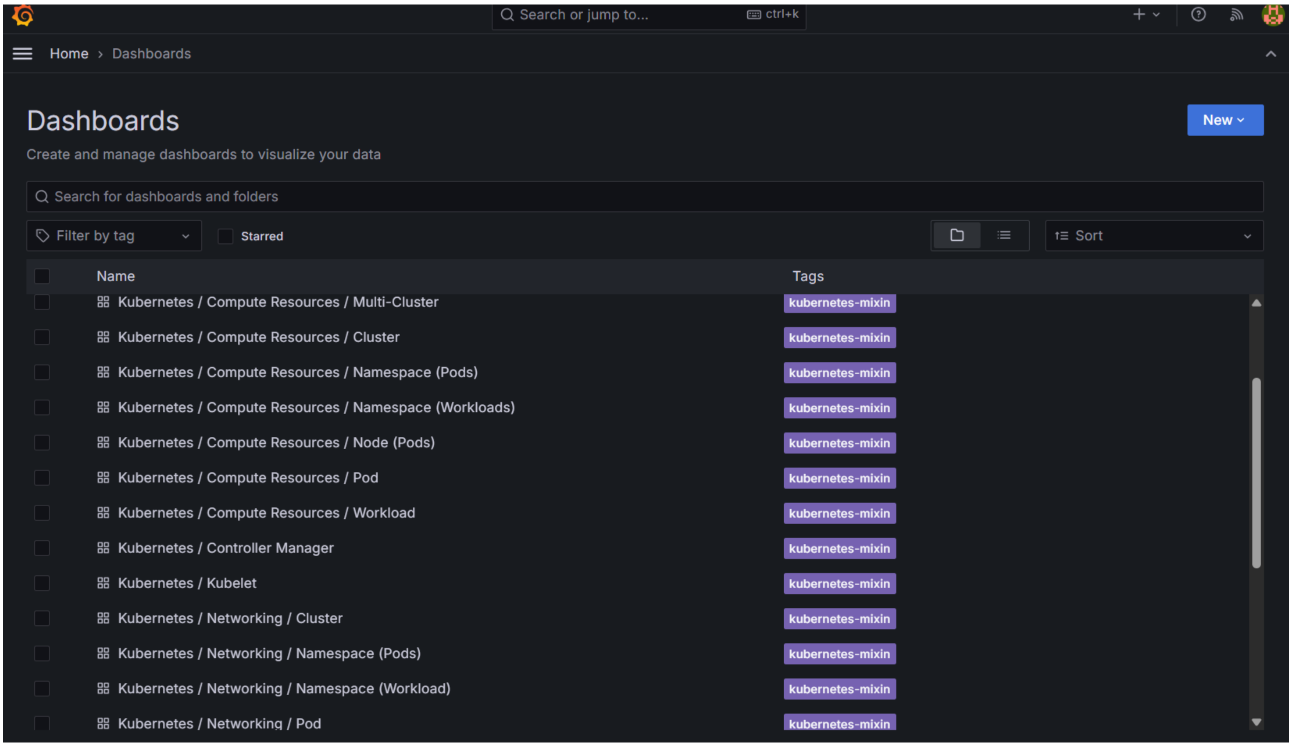
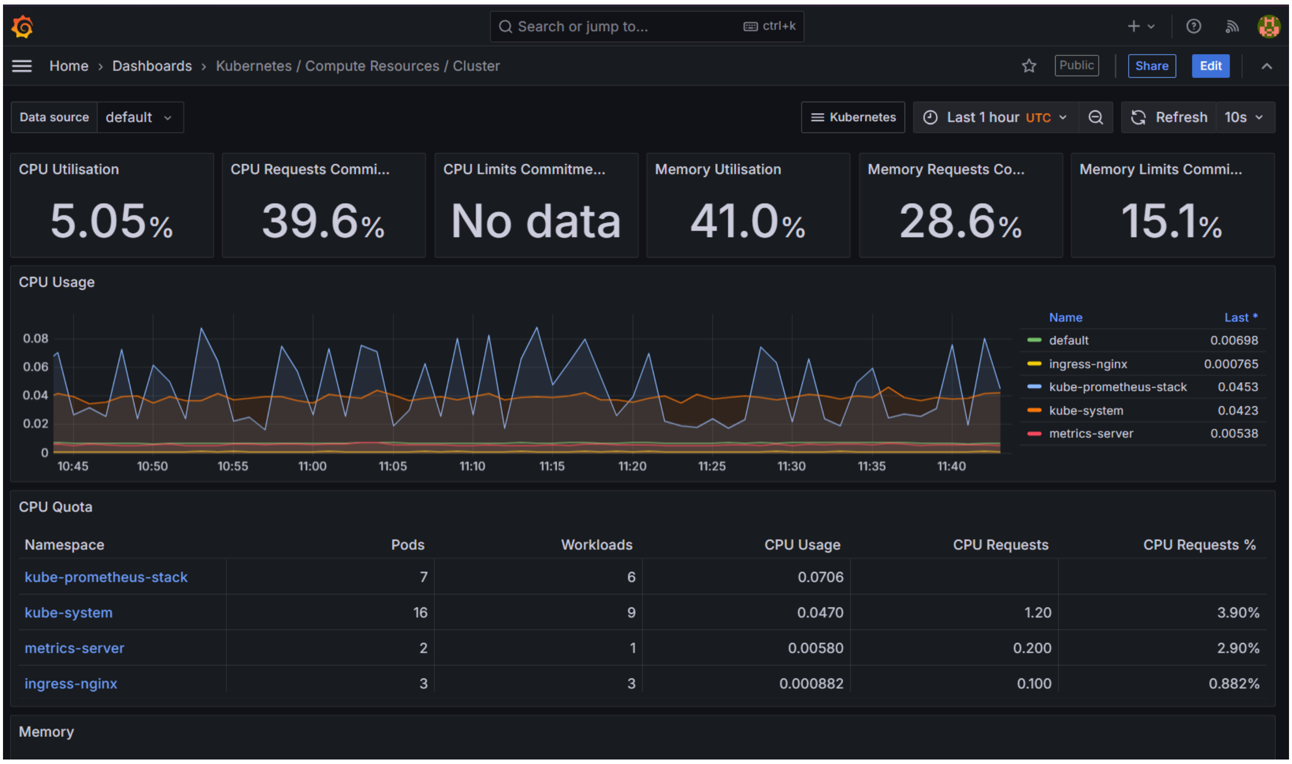


Figure 28 - Delete Integration/Sales

6. Observability

Deploy observability infrastructure within the Kubernetes cluster to monitor the health, performance, and availability of all services using Prometheus and Grafana





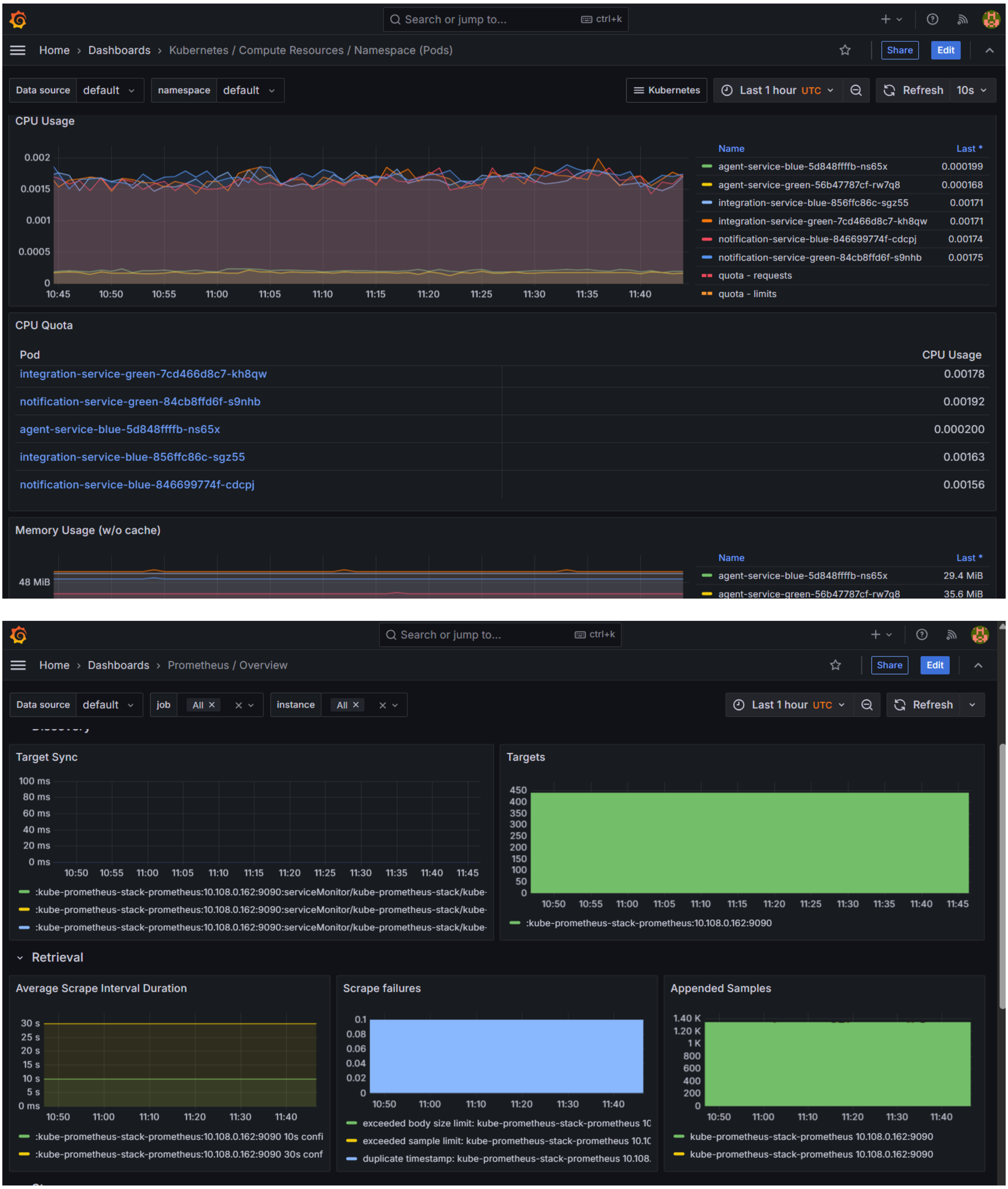


Figure 29 - Grafana-Dashboards

7. MoonInsurance System Deployment Runbook

7.1. Overview

The system comprises multiple microservices deployed in a DigitalOcean Kubernetes environment with CI/CD integration via GitHub Actions. It includes service containerization, automatic deployments, observability using Grafana, and visual analytics via AWS QuickSight.

Runbook name	**MediTrack System Deployment Runbook**
Runbook description	**Runbook to demonstrate the deployment and testing of the MoonInsurance system**, which includes the following core components: 1. **Agent Microservice**: Manages operations related to insurance agents, including adding, updating, retrieving, and removing agent records. This service plays a vital role in maintaining accurate agent profiles across branches. 2. **Integration Microservice**: Handles all insurance product sales transactions, associating agents with products sold, sale dates, and revenue. It ensures accurate tracking of performance and branch-wise sales activity. 3. **Notification Microservice**: Responsible for generating and delivering timely sales performance notifications to agents, helping stakeholders stay informed of milestones, targets, and achievements. 4. **Aggregator Microservice (Scheduled Job)**: Periodically aggregates data across services to calculate team performance, sales against targets, and branch-level metrics. It uploads this data to AWS Redshift, enabling visual analytics via AWS QuickSight.
Owner	Fouzul Hassan
Version	v1.0
Version date	16-04-2025
On this page	- Prerequisites - Deployment steps - Testing procedures - Troubleshooting

7.2. Support Contacts

Expertise Level	Team	Contact
Developer and Owner	Fouzul Hassan	fouzul.20233214@iit.ac.lk

7.3. 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.

8. Dashboard – MoonInsurance System

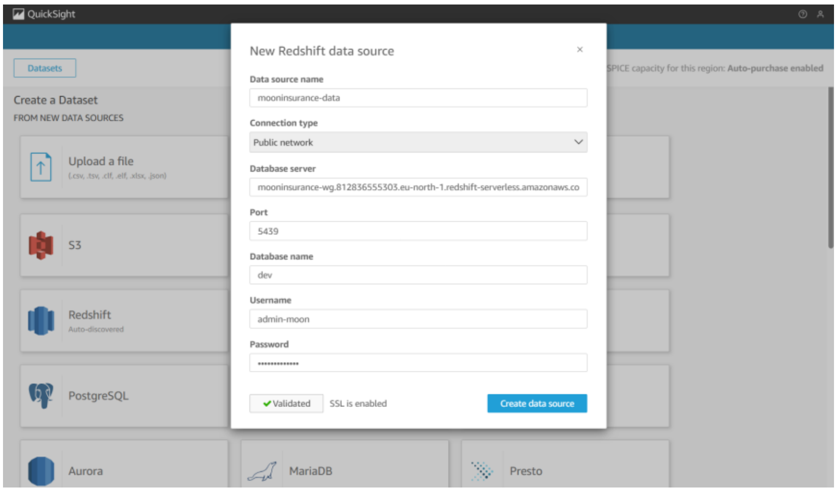


Figure 30 - Connecting-MoonInsurance

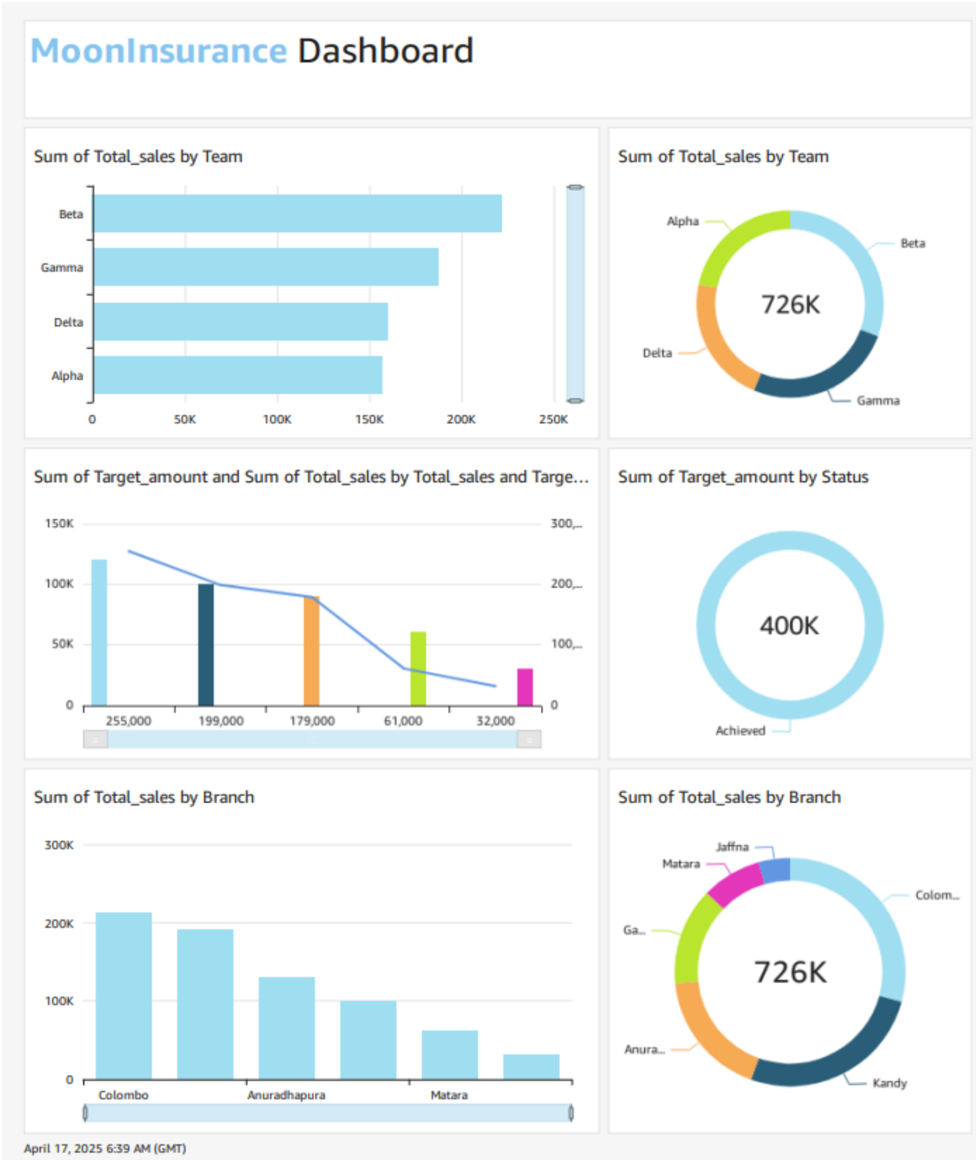


Figure 31 – Dashboard

9. Metrics and High-Performance Analytics Support

Three Key Metrics Considered:

1. Total Sales by Team
2. Total Sales by Branch
3. Target Amount vs. Total Sales by Product

Analytics & Visualization Benefits:

- These metrics are **stored in AWS Redshift**, enabling **scalable, high-performance analytics** across large volumes of sales data.
- Visualized via **AWS QuickSight**, these interactive dashboards allow administrators to:
    - **Identify top-performing teams and branches**
    - **Monitor progress against product targets**
    - **Gain actionable insights** to adjust sales strategies and resource allocation
- The **real-time dashboards** enhance operational decision-making and support strategic planning for insurance corporate providers.