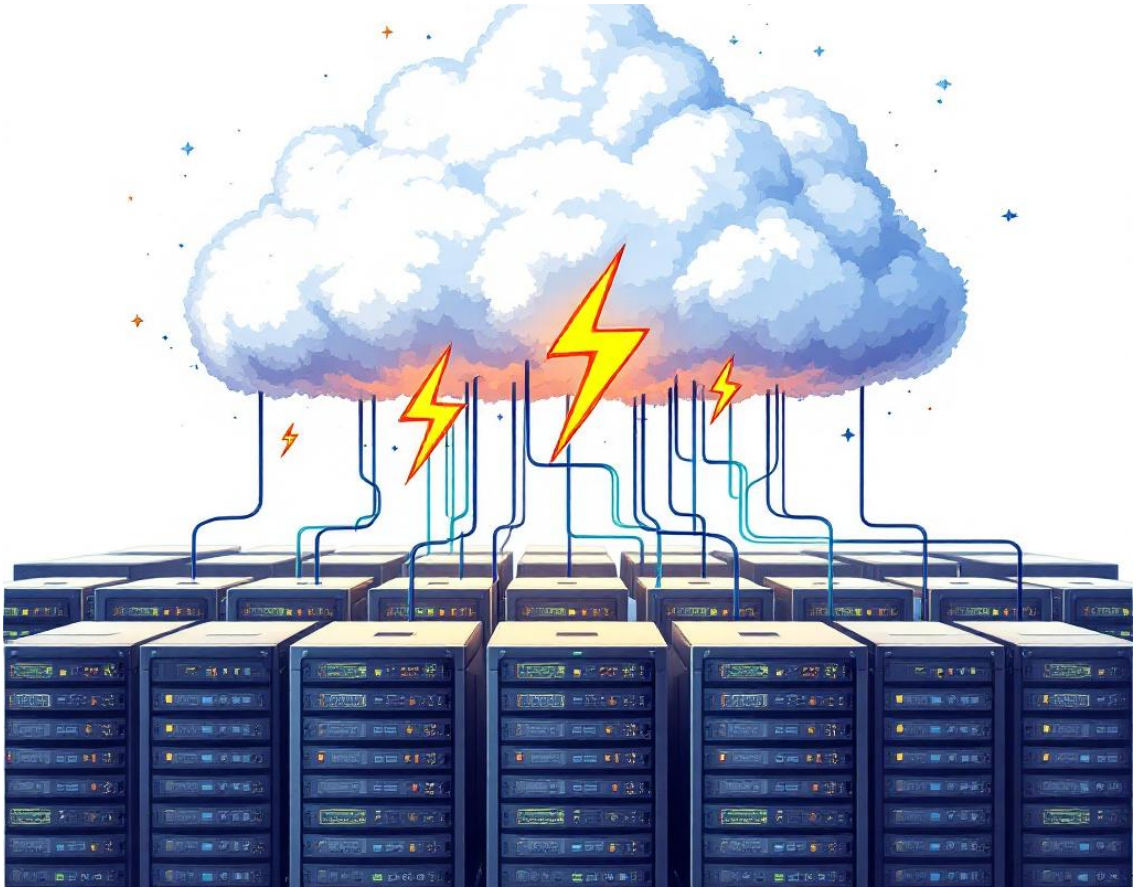
ROBERT GORDON UNIVERSITY ABERDEEN

CMM707 - Cloud Computing

Moon Agent Tracker – Coursework Report

Cloud-based Microservices Solution



Student ID: [Enter Student ID]

Date: [Enter Date]

1. Introduction

Moon Insurance, a growing insurance company, identified the need for a digital platform to manage and analyze its agent activities, sales transactions, and team performance across multiple branches. The objective was to build a cloud-native microservices application that is scalable, secure, fault-tolerant, and cost-effective. The outcome was Moon Agent Tracker — a Kubernetes-based solution that transitioned from local development using Minikube to a production-grade deployment on AWS Elastic Kubernetes Service (EKS).

This report presents my development journey, from early mistakes to final optimization. It includes architecture, automation, observability, security, and ethical design considerations — all tailored to industry best practices and academic requirements.

2. Development Journey & Deployment Strategy

Phase 1: Local Testing with Minikube

My cloud-native learning journey began with a Minikube setup, where I containerized the main three microservices — *Agent*, *Integration*, *Notification* (*Aggregator cron-job* added later under EKS). The initial hurdles included **misconfigured ports**, **relying on localhost for inter-service calls instead of Kubernetes DNS**, and **forgetting to expose service ports via Cluster IP**.

To resolve these, I created a Kubernetes namespace *cmm707-microservices*, used service discovery by DNS (*http://<service-name>:port*), and tested services using kubectl port-forward. I built and loaded Docker images into the local cluster and configured .env files through *ConfigMaps* and *Secrets*.

Health endpoints (/health) were added to all services, laying the foundation for CI test hooks and observability.

Phase 2: AWS EKS Setup

I initially tried to configure AWS EKS through the AWS Console, which proved error-prone due to misunderstood IAM permissions and manual missteps. Eventually, I adopted **eksctl** for declarative provisioning. I created a **3-node cluster** in the **ap-southeast-1** region:

After configuring kubectl with the EKS context, I deployed the same manifests from my local setup, adjusted for cloud networking. Exposing services using LoadBalancer and assigning Elastic IPs ensured stable endpoints.

Using the list of commands, I created the cluster & confirmed external connectivity, and adjusted health checks to match cloud latency expectations.

```
eksctl create cluster \
    --name moon-cluster \
    --region ap-southeast-1 \
    --nodegroup-name standard-workers \
    --node-type t3.medium \
    --nodes 2 \
    --nodes-min 1 \
    --managed
```
Create an EKS Cluster

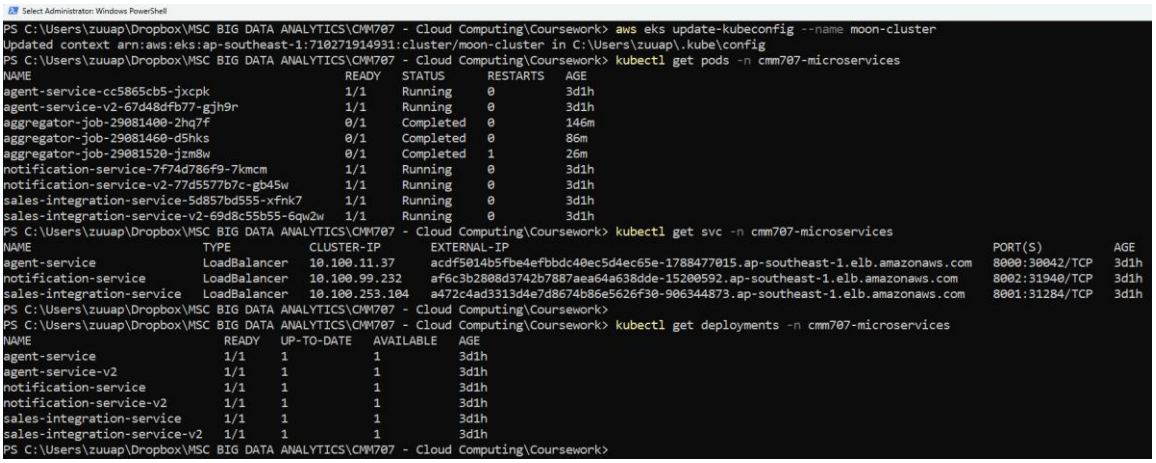
`aws eks update-kubeconfig --name moon-cluster` **Connecting to EKS Cluster**

`kubectl get pods -n cmm707-microservices` **List All Namespaces**

`kubectl get svc -n cmm707-microservices` **List All Pods in the Namespaces**

`kubectl get deployments -n cmm707-microservices` **List All Services**

`kubectl describe svc service-name -n cmm707-microservices` **Describe Each Service**



```
Select Administrator: Windows PowerShell
PS C:\Users\zuuap\Dropbox\MSC BIG DATA ANALYTICS\CMM707 - Cloud Computing\Coursework> aws eks update-kubeconfig --name moon-cluster
Updated context arn:aws:eks:ap-southeast-1:710271914931:cluster/moon-cluster in C:\Users\zuuap\.kube\config
PS C:\Users\zuuap\Dropbox\MSC BIG DATA ANALYTICS\CMM707 - Cloud Computing\Coursework> kubectl get pods -n cmm707-microservices
NAME                                        READY   STATUS      RESTARTS   AGE
agent-service-cc5865cb5-jxcpk               1/1     Running     0          3d1h
agent-service-v2-67d48dfb77-gjh9r           1/1     Running     0          3d1h
aggregator-job-29081400-2hq7f               0/1     Completed   0          146m
aggregator-job-29081460-d5hks               0/1     Completed   0          86m
aggregator-job-29081520-jzm8w               0/1     Completed   1          26m
notification-service-7f74d786f9-7kmcm       1/1     Running     0          3d1h
notification-service-v2-77d5577b7c-gb45w    1/1     Running     0          3d1h
sales-integration-service-5d857bd555-xfnk7  1/1     Running     0          3d1h
sales-integration-service-v2-69d8c55b55-6qw2w   1/1     Running     0          3d1h
PS C:\Users\zuuap\Dropbox\MSC BIG DATA ANALYTICS\CMM707 - Cloud Computing\Coursework> kubectl get svc -n cmm707-microservices
NAME                    TYPE           CLUSTER-IP       EXTERNAL-IP                                                                PORT(S)          AGE
agent-service           LoadBalancer   10.100.11.37     acdf5014b5fbe4efbbdc40ec5d4ec65e-1788477015.ap-southeast-1.elb.amazonaws.com   8000:30042/TCP   3d1h
notification-service    LoadBalancer   10.100.99.232    af6c3b2808d3742b7887aea64a638dde-15200592.ap-southeast-1.elb.amazonaws.com     8002:31940/TCP   3d1h
sales-integration-service   LoadBalancer   10.100.253.104   a472c4ad3313d4e7d8674b86e5626f30-906344873.ap-southeast-1.elb.amazonaws.com    8001:31284/TCP   3d1h
PS C:\Users\zuuap\Dropbox\MSC BIG DATA ANALYTICS\CMM707 - Cloud Computing\Coursework>
PS C:\Users\zuuap\Dropbox\MSC BIG DATA ANALYTICS\CMM707 - Cloud Computing\Coursework> kubectl get deployments -n cmm707-microservices
NAME                        READY   UP-TO-DATE   AVAILABLE   AGE
agent-service               1/1     1            1           3d1h
agent-service-v2            1/1     1            1           3d1h
notification-service        1/1     1            1           3d1h
notification-service-v2     1/1     1            1           3d1h
sales-integration-service   1/1     1            1           3d1h
sales-integration-service-v2   1/1     1            1           3d1h
PS C:\Users\zuuap\Dropbox\MSC BIG DATA ANALYTICS\CMM707 - Cloud Computing\Coursework>
```

```
PS C:\Users\zuuap\Dropbox\MSC BIG DATA ANALYTICS\CMM707 - Cloud Computing\Coursework> kubectl describe svc agent-service -n cmm707-microservices
Name:                     agent-service
Namespace:                cmm707-microservices
Labels:                   <none>
Annotations:              <none>
Selector:                 app=agent-service,version=v2
Type:                     LoadBalancer
IP Family Policy:         SingleStack
IP Families:              IPv4
IP:                       10.100.11.37
IPs:                      10.100.11.37
LoadBalancer Ingress:     acdf5014b5fbe4efbbdc40ec5d4ec65e-1788477015.ap-southeast-1.elb.amazonaws.com
Port:                     <unset>  8000/TCP
TargetPort:               8000/TCP
NodePort:                 <unset>  30042/TCP
Endpoints:                192.168.55.6:8000
Session Affinity:         None
External Traffic Policy:  Cluster
Internal Traffic Policy:  Cluster
Events:                   <none>
```

```
PS C:\Users\zuuap\Dropbox\MSC BIG DATA ANALYTICS\CMM707 - Cloud Computing\Coursework> kubectl describe svc notification-service -n cmm707-microservices
Name:                     notification-service
Namespace:                cmm707-microservices
Labels:                   <none>
Annotations:              <none>
Selector:                 app=notification-service,version=v2
Type:                     LoadBalancer
IP Family Policy:         SingleStack
IP Families:              IPv4
IP:                       10.100.99.232
IPs:                      10.100.99.232
LoadBalancer Ingress:     af6c3b2808d3742b7887aea64a638dde-15200592.ap-southeast-1.elb.amazonaws.com
Port:                     <unset>  8002/TCP
TargetPort:               8002/TCP
NodePort:                 <unset>  31940/TCP
Endpoints:                192.168.59.75:8002
Session Affinity:         None
External Traffic Policy:  Cluster
Internal Traffic Policy:  Cluster
Events:                   <none>
```

```
PS C:\Users\zuuap\Dropbox\MSC BIG DATA ANALYTICS\CMM707 - Cloud Computing\Coursework> kubectl describe svc sales-integration-service -n cmm707-microservices
Name:                     sales-integration-service
Namespace:                cmm707-microservices
Labels:                   <none>
Annotations:              <none>
Selector:                 app=sales-integration-service,version=v2
Type:                     LoadBalancer
IP Family Policy:         SingleStack
IP Families:              IPv4
IP:                       10.100.253.104
IPs:                      10.100.253.104
LoadBalancer Ingress:     a472c4ad3313d4e7d8674b86e5626f30-906344873.ap-southeast-1.elb.amazonaws.com
Port:                     <unset>  8001/TCP
TargetPort:               8001/TCP
NodePort:                 <unset>  31284/TCP
Endpoints:                192.168.2.52:8001
Session Affinity:         None
External Traffic Policy:  Cluster
Internal Traffic Policy:  Cluster
Events:                   <none>
```

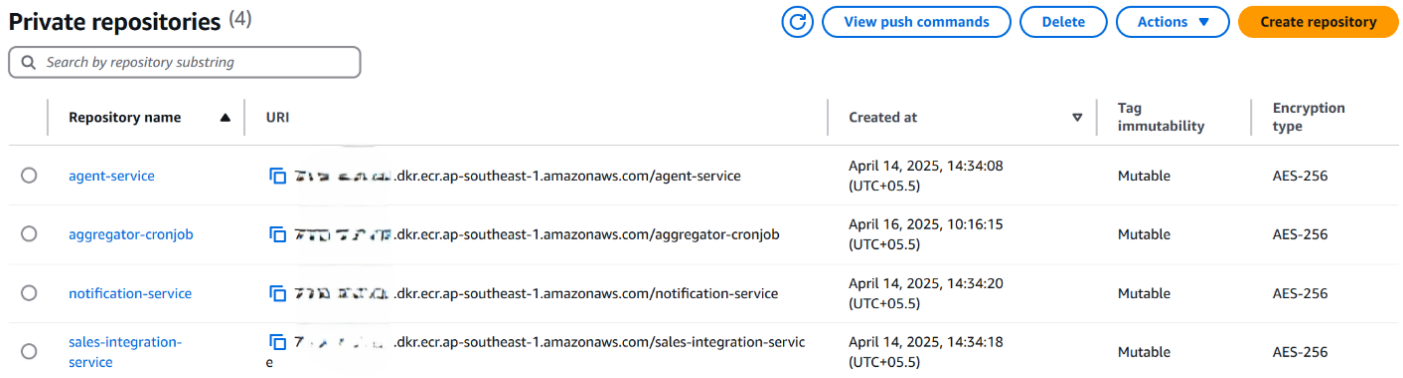
3. Microservices Architecture

3.1 Services Breakdown

While microservices were given for this solution, the key challenge was in defining clear service boundaries and avoiding redundancy or unnecessary coupling. My initial implementation blurred responsibilities across services, making debugging and scaling more difficult. Through iteration and feedback, I refactored architecture to embrace the **single-responsibility principle** more rigorously, ensuring each service managed a distinct concern in the overall sales lifecycle. This restructuring improved maintainability enabled targeted testing, and simplified CI/CD workflows.

Service	Function
Agent Service	Handles agent profiles, product authorization, and CRUD operations.
Integration	Listens for sales data pushes from core systems and logs transaction entries.
Notification	Monitors sales target, send alerts via email or API when thresholds are met.
Aggregator	A scheduled job to compute KPIs (sales, branches, teams) and write to Redshift.

Each microservice is independently deployed using a *Deployment + Service YAML* (same file), and environmental configs are injected via *ConfigMaps and Secrets*. Below I have attached the AWS Console Dashboard View for AWS Elastic Container Registry (ECR), AWS Elastic Kubernetes Service (EKS).



Private repositories (4)

View push commands | Delete | Actions | Create repository

Search by repository substring

Repository name	URI	Created at	Tag immutability	Encryption type
agent-service	.dkr.ecr.ap-southeast-1.amazonaws.com/agent-service	April 14, 2025, 14:34:08 (UTC+05.5)	Mutable	AES-256
aggregator-cronjob	.dkr.ecr.ap-southeast-1.amazonaws.com/aggregator-cronjob	April 16, 2025, 10:16:15 (UTC+05.5)	Mutable	AES-256
notification-service	.dkr.ecr.ap-southeast-1.amazonaws.com/notification-service	April 14, 2025, 14:34:20 (UTC+05.5)	Mutable	AES-256
sales-integration-service	.dkr.ecr.ap-southeast-1.amazonaws.com/sales-integration-service	April 14, 2025, 14:34:18 (UTC+05.5)	Mutable	AES-256

3.2 Directory Structure

- The k8s/ directory consolidates all Kubernetes resource definitions required *for deploying the microservices and managing the environment*. Each API-driven microservice (Agent, Integration, Notification) is configured with two deployment files — v1 (Blue) and v2 (Green) to support Blue-Green deployment strategy. Other files like *aggregator-cronjob.yaml*, *configmap.yaml*, and *secret.yaml* define job scheduling, environment-specific variables, and credentials, respectively. Shared infrastructure resources such as the namespace and ingress routing rules are also included here for consistency across environments.

```
k8s/
├── agent-service.yaml                # Agent Service v1
├── agent-service-v2.yaml             # Agent Service v2
├── sales-integration-service.yaml    # Integration Service v1
├── sales-integration-service-v2.yaml # Integration Service v2
├── notification-service.yaml         # Notification Service v1
├── notification-service-v2.yaml      # Notification Service v2
├── aggregator-cronjob.yaml           # Aggregator as CronJob
├── configmap.yaml                    # Environment variables and config
├── secret.yaml                       # DB credentials and other secrets
├── namespace.yaml                    # Namespace definition
├── ingress.yaml                      # Ingress routing rules
```

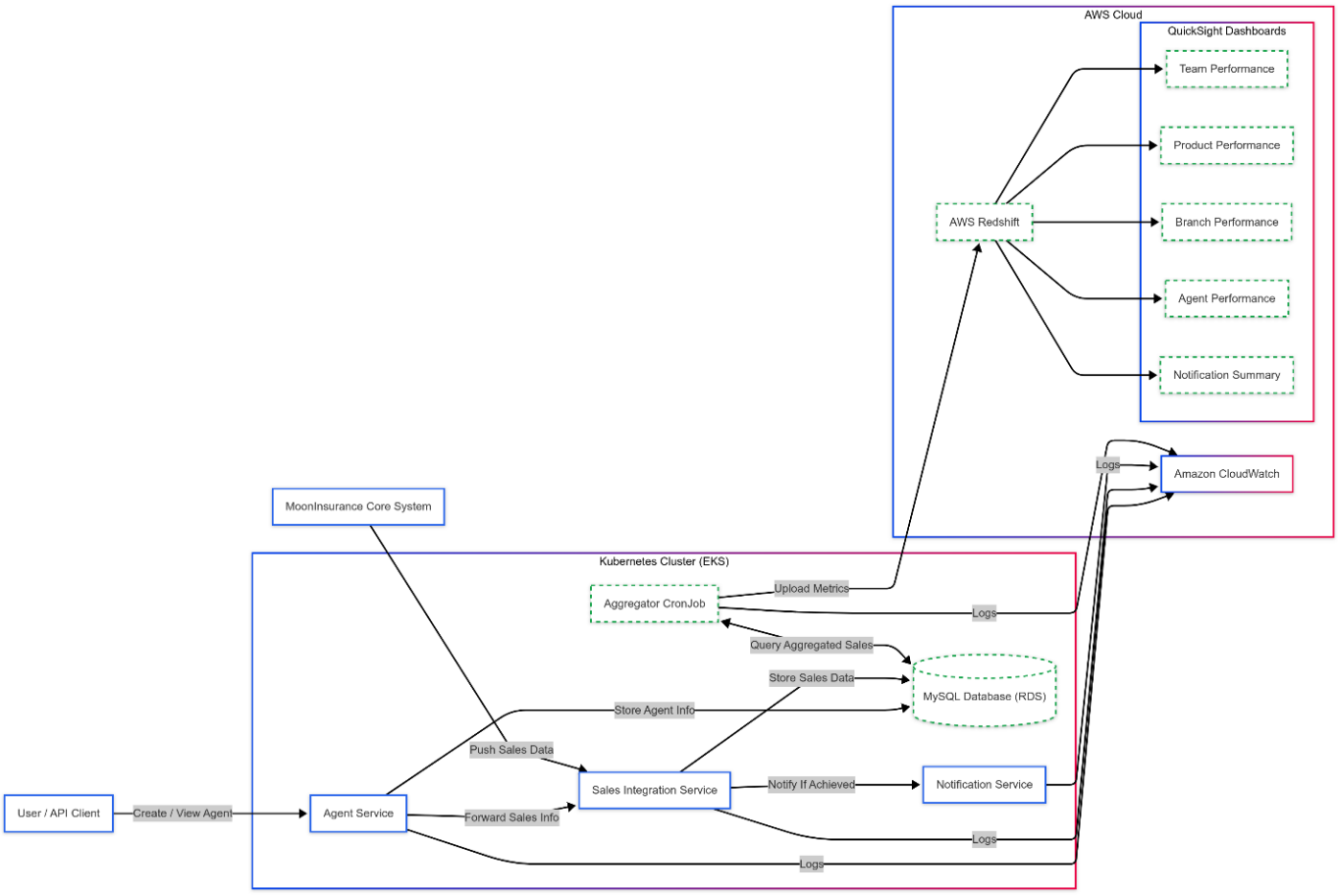
- The following structure represents the organized layout of all four microservices in the Moon Agent Tracker solution. Each service is independently containerized and follows a modular approach with clearly defined components for *application logic* (app/), *dependencies* (requirements.txt), *Docker configuration*, and *Kubernetes deployment files* (k8s/). To support zero-downtime rollouts, API-based services (Agent, Integration, Notification) are deployed using **Blue-Green deployment patterns**, represented as separate v1 and v2 YAML files. The Aggregator service is designed as a Kubernetes Cronjob to periodically (hourly basis) compute and upload performance data.

```
microservices/
├── agent-service/
│   ├── app/
│   │   ├── init.py
│   │   ├── database.py
│   │   ├── main.py
│   │   ├── models.py
│   │   ├── routes.py
│   │   └── schemas.py
│   ├── Dockerfile
│   ├── requirements.txt
│   └── k8s/
│       ├── agent-service.yaml          # v1 (Blue)
│       └── agent-service-v2.yaml       # v2 (Green)
│
├── sales-integration-service/
│   ├── app/
│   │   ├── init.py
│   │   ├── database.py
│   │   ├── main.py
│   │   ├── models.py
│   │   ├── routes.py
│   │   └── schemas.py
│   ├── Dockerfile
│   ├── requirements.txt
│   └── k8s/
│       ├── sales-integration-service.yaml          # v1 (Blue)
│       └── sales-integration-service-v2.yaml       # v2 (Green)
```

```
├── notification-service/
│   ├── app/
│   │   ├── init.py
│   │   ├── database.py
│   │   ├── main.py
│   │   ├── models.py
│   │   ├── routes.py
│   │   └── schemas.py
│   ├── Dockerfile
│   ├── requirements.txt
│   └── k8s/
│       ├── notification-service.yaml        # v1 (Blue)
│       └── notification-service-v2.yaml     # v2 (Green)
│
└── aggregator-service/
    ├── app/
    │   ├── init.py
    │   └── aggregator.py
    ├── Dockerfile
    ├── requirements.txt
    └── k8s/
        └── aggregator-cronjob.yaml          # Scheduled batch job
```

3.3 Solution Architecture Design

Below is the *Solution Architecture Design* illustrating the interaction between microservices deployed on **AWS EKS (Elastic Kubernetes Service)**, core business systems, and AWS-native services used for analytics, monitoring, and persistence.



3.3.1 Core Components and Data Flow

1. User / API Client

- Interacts with the system by creating or viewing agent records through the Agent Service.

2. Agent Service

- Manage agent-related data and forward valid sales info to the Sales Integration Service.
- Also receives agent context from the Moon Insurance Core System.

3. Sales Integration Service

- Receives live sales data and stores it in the MySQL Database (RDS).
- It checks conditions and, when targets are met, notifies the Notification Service.

4. Notification Service

- Send out alerts when team or individual targets are achieved.
- Sends logs to Amazon CloudWatch for observability.

5. Aggregator Cronjob

- Periodically queries the MySQL database to aggregate KPIs.
- Uploads aggregated data into AWS Redshift.

3.3.2 Cloud-Integrated Layers

1. Data Layer

- MySQL (RDS): Stores agents and sales data persistently.
- AWS Redshift: Centralized data warehouse for analytical queries and dashboards.

2. Analytics Layer

- Amazon QuickSight dashboards are powered by Redshift and show:
    - Team Performance
    - Product Performance
    - Branch Performance
    - Agent Performance
    - Notification Summary

3. Monitoring Layer

- All microservices stream logs to Amazon CloudWatch for diagnostics, auditing, and performance monitoring.

4. CI/CD Pipeline & Blue-Green Deployment

Initially, my CI/CD workflow focused purely on deployment — it lacked integration tests and basic validation steps. This caused several broken builds to reach production, prompting a redesigning of the pipeline.

The updated CI/CD pipeline is now robust, fully automated, and tightly integrated with health validation logic. It is triggered on every push to the main branch and supports seamless Blue-Green deployments for all three API-based microservices (Agent, Integration, Notification).

4.1 CI/CD Workflow Steps:

1. **Trigger**: Any commit to the main branch initiates the workflow.

```yaml
on:
  push:
    branches:
      - main
  schedule:
    - cron: '0 6 * * *'
```

2. **Build & Package**: Docker images are built for each microservice and tagged with the current commit hash.

```yaml
- name: Build Docker images
  run: |
    docker build -t agent-service:${{ github.sha }} ./agent-service
    docker build -t sales-integration-service:${{ github.sha }} ./sales-integration-service
    docker build -t notification-service:${{ github.sha }} ./notification-service
```

3. **Push to AWS ECR**: Built images are pushed to respective Elastic Container Registry (ECR) repositories.

```yaml
- name: Push Docker images to ECR
  run: |
    aws ecr get-login-password --region ap-southeast-1 | docker login --username AWS --password-stdin <your-ecr-url>
    docker tag agent-service:${{ github.sha }} <your-ecr-url>/agent-service:${{ github.sha }}
    docker push <your-ecr-url>/agent-service:${{ github.sha }}
```

4. **Deploy to EKS**: Kubernetes manifests for v2 (Green) are applied using kubectl, running in the GitHub Actions runner.

```
- name: Deploy Blue (v2) versions of services
  run: |
    kubectl apply -f k8s/agent-service-v2.yaml
    kubectl apply -f k8s/sales-integration-service-v2.yaml
    kubectl apply -f k8s/notification-service-v2.yaml
```

5. **Health Validation**: Each service is port-forwarded and tested using a custom Python script to validate /health endpoints.

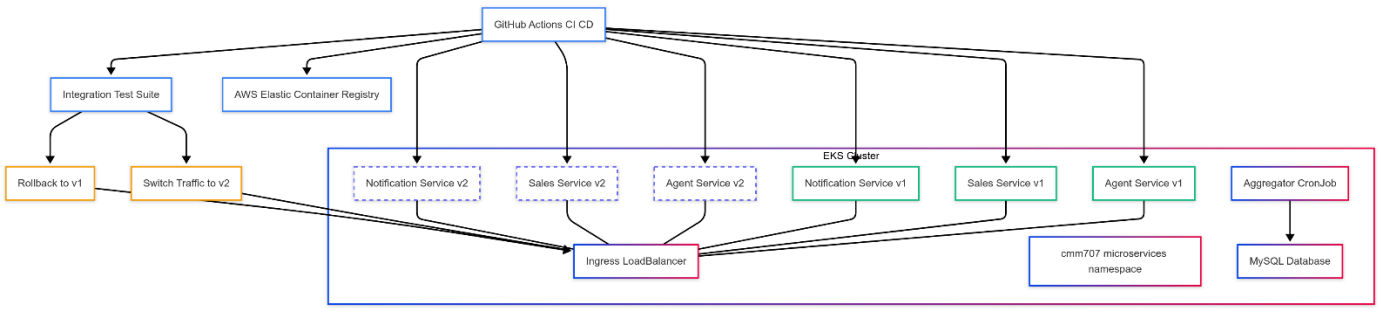
```
- name: Run integration tests on EKS
  run: |
    pip install requests
    python scripts/run-tests-eks.py
```

6. **Traffic Switch**: Only upon successful health checks, traffic is routed to the v2 pods using a shared Kubernetes Service object.

```
- name: Switch traffic to v2 (Blue-Green switch)
  run: |
    kubectl patch service agent-service -n cmm707-microservices -p '{"spec": {"selector": {"app": "agent-service", "version": "v2"}}}'
    kubectl patch service sales-integration-service -n cmm707-microservices -p '{"spec": {"selector": {"app": "sales-integration-service", "version": "v2"}}}'
    kubectl patch service notification-service -n cmm707-microservices -p '{"spec": {"selector": {"app": "notification-service", "version": "v2"}}}'

- name: Rollback to v1 if tests fail
  if: failure()
  run: |
    echo "Tests failed. Rolling back to version v1..."
    kubectl patch service agent-service -n cmm707-microservices -p '{"spec": {"selector": {"app": "agent-service", "version": "v1"}}}'
    kubectl patch service sales-integration-service -n cmm707-microservices -p '{"spec": {"selector": {"app": "sales-integration-service", "version": "v1"}}}'
    kubectl patch service notification-service -n cmm707-microservices -p '{"spec": {"selector": {"app": "notification-service", "version": "v1"}}}'
```

This diagram illustrates the automated continuous deployment pipeline powered by **GitHub Actions**, which orchestrates versioned microservice rollouts into an Amazon EKS.



4.2 Blue-Green Deployment Logic

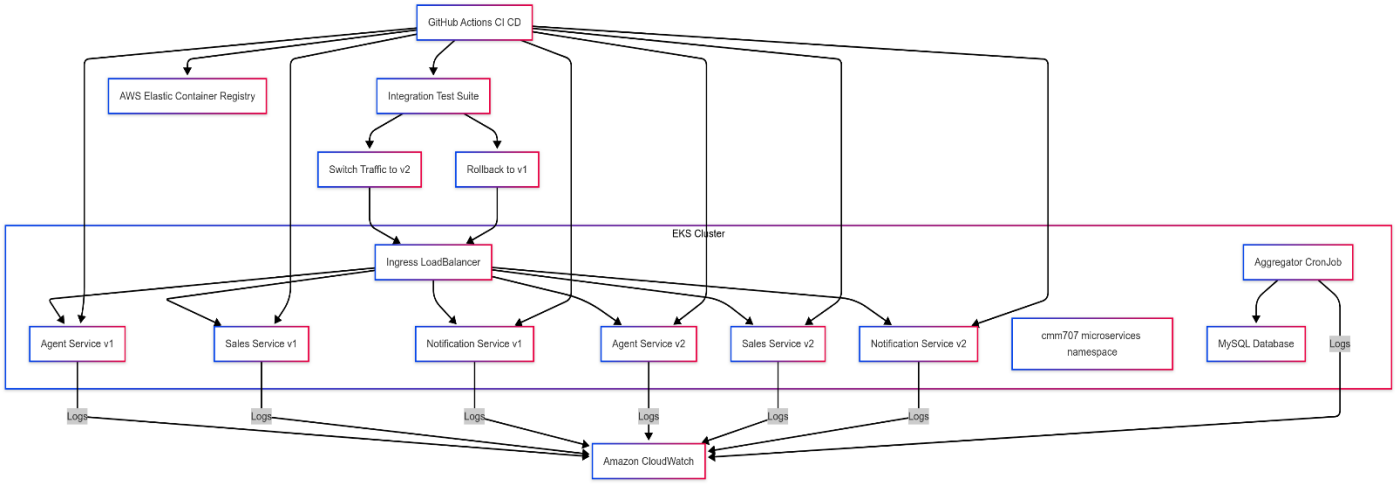
Each deployment is split into two versions:

- v1 (Blue): Current live version
- v2 (Green): New candidate version

This architecture diagram illustrates how your microservices are deployed, versioned, validated, and monitored inside an AWS EKS environment using a **Blue-Green deployment strategy** integrated with **GitHub Actions CI/CD**.

After deploying v2 versions of services; an **Integration Test Suite** is executed:

- If **tests pass** → traffic is switched to v2 via the **Ingress LoadBalancer**.
- If **tests fail** → rollback is triggered to retain v1 as the live version.



5. Observability and Monitoring

During early deployment and testing phases, identifying issues like pod failures and performance bottlenecks was challenging due to the lack of centralized logging and monitoring. To overcome this, I integrated Amazon CloudWatch as the primary observability solution for the entire Kubernetes cluster and microservices.

All services — including the Agent, Sales Integration, Notification, and Aggregator — are configured to stream logs directly to Amazon CloudWatch Logs. This was achieved using built-in AWS EKS logging support and Kubernetes-native logging mechanisms.

🧩 Key features of the setup:

- Logs from containers are automatically collected and routed to individual log streams.
- Structured logs include timestamps, log levels, and service-specific tags for filtering.
- Cronjobs (like the Aggregator) also output logs to CloudWatch for historical execution tracking.

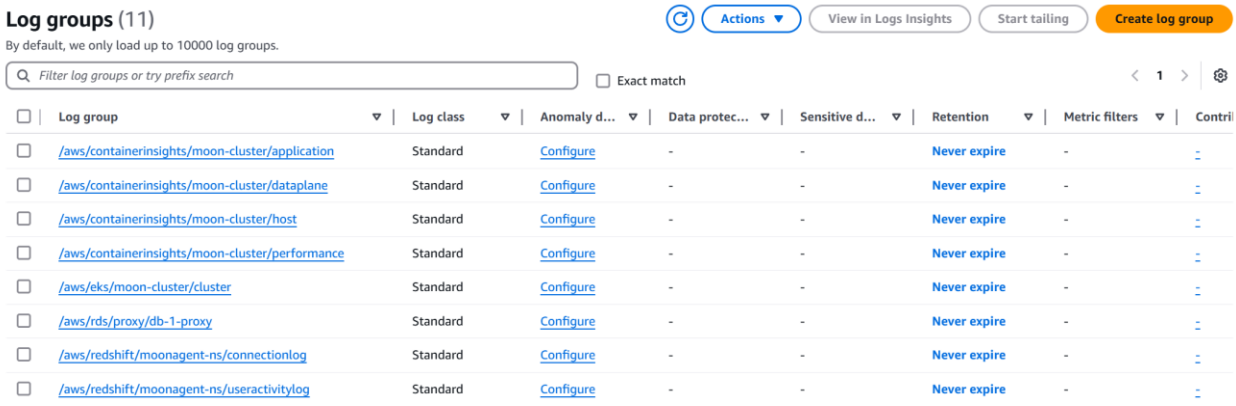
🔍 What Was Monitored

- Pod status and lifecycle events (e.g., restarts, terminations, crash loops)
- Application-level logs from /main.py, background tasks, and health checks
- Cronjob executions (aggregator logs) and errors from Redshift or MySQL interactions
- Service availability through periodic kubectl get checks and health logs.

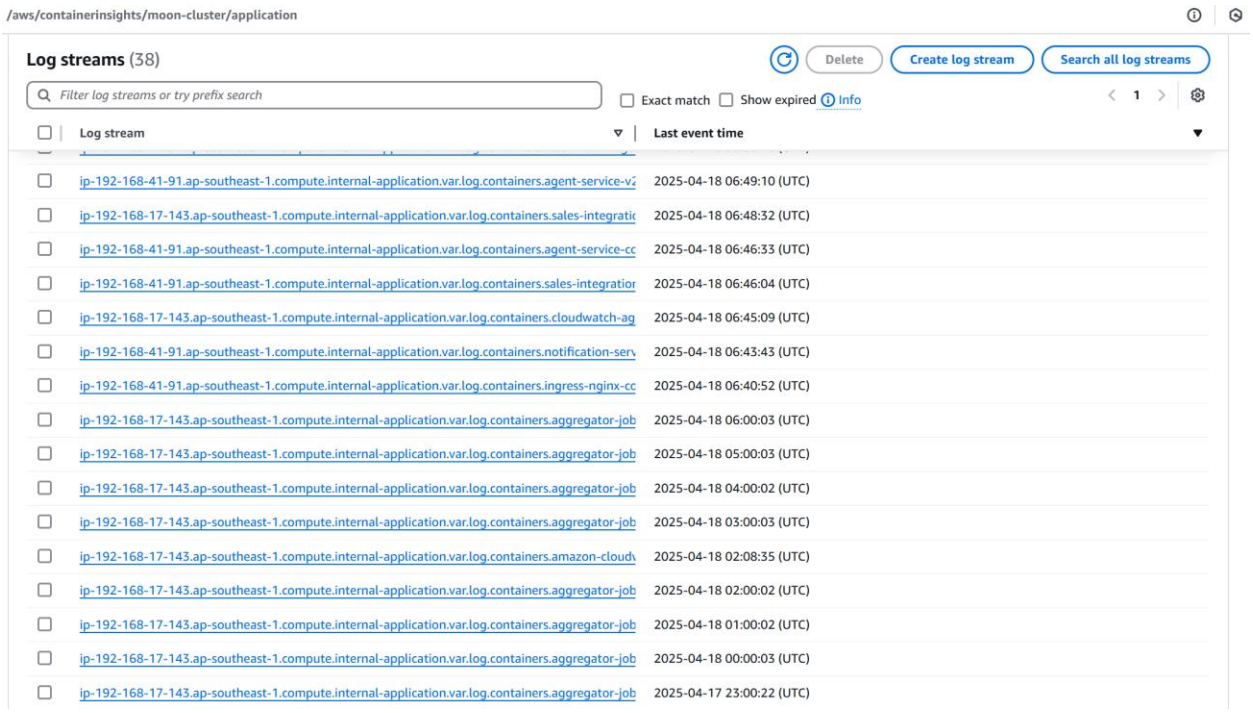
📊 Benefits of CloudWatch

- Centralized visibility into logs across all microservices and namespaces
- Real-time diagnostics and troubleshooting for production and development environments.
- No additional observability stack required (like Prometheus/Grafana), reducing setup complexity.
- Supports IAM-based access control for secure audit trails and logging.

Each entry in this table represents a **log group** – a collection of logs from a specific component or service in your AWS environment. These are organized by AWS namespaces and services.



Each log group consists of log stream listed corresponds to a **container** running inside your **EKS cluster**, and includes services like:



6. Aggregation & Track Metrics

6.1 Aggregator Cronjob (Kubernetes)

Initially, metric aggregation was handled by a continuously running API endpoint. This approach consumed unnecessary memory and left background processing unscalable. To improve efficiency and reliability, I transitioned the solution to a Kubernetes Cronjob that executes **once every hour**. This model ensures the system remains stateless and cost-effective while maintaining up-to-date analytics in near real-time.

The Cronjob performs the following tasks:

- Connects to the **MySQL RDS** database to fetch raw sales data.
- Aggregates performance by **team**, **product**, and **branch**
- Uploads summarized data to **AWS Redshift** using the psycopg2 Python library.
- Logs execution metrics and completion status to **Amazon CloudWatch**

6.2 AWS Redshift Schema Design

Sales performance data is structured in Redshift using dedicated tables optimized for dashboard queries.

```
-- Insight 1: Best Performing Sales Teams
CREATE TABLE IF NOT EXISTS redshift_team_performance (
    team_name VARCHAR(255),
    total_sales NUMERIC,
    aggregation_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

```sql
-- Insight 2: Products That Achieved Sales Targets
CREATE TABLE IF NOT EXISTS redshift_product_performance (
    product_name VARCHAR(255),
    total_sales NUMERIC,
    status VARCHAR(50),  -- 'Achieved' or 'Not Achieved'
    aggregation_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Insight 3: Branch-wise Sales Performance
CREATE TABLE IF NOT EXISTS redshift_branch_performance (
    branch_name VARCHAR(255),
    total_sales NUMERIC,
    aggregation_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Insight 4: Most Notified Agents
CREATE TABLE IF NOT EXISTS redshift_agent_notifications (
    agent_name VARCHAR(255),
    notification_count INTEGER,
    aggregation_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Insight 5: Notification Status Summary
CREATE TABLE IF NOT EXISTS redshift_notification_summary (
    status VARCHAR(50), -- e.g., 'PENDING', 'SENT'
    notification_count INTEGER,
    aggregation_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

7. Business Intelligence & Quicksight Dashboards

To convert backend sales and performance metrics into business-friendly visual insights, I used **Amazon QuickSight** to connect directly to the AWS Redshift data warehouse. This allowed the creation of dynamic dashboards tailored to the key insights produced by the Aggregator Cronjob.

7.1 Data Source Integration

- Connected QuickSight to the Redshift cluster using an **IAM role with federated access.**
- Selected the five main tables created by the Cronjob:
  - redshift_team_performance
  - redshift_product_performance
  - redshift_branch_performance
  - redshift_agent_notifications
  - redshift_notification_summary

7.2 Dashboard Features

Each dashboard was designed with **interactive filters** and **custom visualizations** to support decision-making across multiple levels of the organization.

Insight Group	Insight Description
Team Performance	Top 5 highest performing teams by total sales (overall)
	Bottom 5 lowest performing branches (last month)
	Daily sales trends over the past 7 days
	Monthly sales growth per team (last 3 months)
Product Performance	Products with the highest target achievement consistency
	Top 3 fastest-growing products (month-over-month growth)
	Daily product performance trends (last month)
Branch Analysis	Top performing branches based on 24-hour sales
	Monthly sales comparison across branches (last 6 months)
	Contribution percentage of each branch to total sales
Agent Notifications	Agents receiving the highest number of notifications (last 24 hours)
	Monthly notification trends per agent (last 3 months)
	Agents with the highest average notifications per record
Notification Delivery Summary	Total notifications sent vs. pending (last month breakdown)
	Daily delivery trends per notification status (last 7 days)
	Overall percentage breakdown of notification statuses (e.g., SENT vs. PENDING)

8. Security & Ethics Considerations

Security was a key area of improvement in my journey. Initially, secrets were hardcoded. I transitioned to Kubernetes Secrets and encrypted sensitive data using AWS KMS.

Security Highlights

- **Kubernetes Secrets for database credentials.**
- **IAM role-based access (IRSA) for Redshift and ECR.**
- **HTTPS enforced via ALB listener rules.**
- **Private subnets used for sensitive services.**

Ethical & Legal Aspects

- **Data pseudonymization applied to all personally identifiable agent data.**
- **Aggregations ensure no single agent is exposed in raw form.**
- **The platform complies with GDPR and RGU's academic integrity policies.**

[Insert Screenshot: IAM role policy attached to aggregator service account]

---

9. Reflection and Conclusion

I started with many gaps in Kubernetes networking and security. Early mistakes like skipped test hooks, wrong service types, and poorly structured YAMLs were corrected through hands-on experimentation and feedback.

Transitioning from Minikube to EKS gave me a true taste of cloud operations, including cost trade-offs, load balancing, persistent IP needs, and scalability. Adding CI/CD, observability, and AWS-native integrations like Redshift and QuickSight turned this project into a mature, enterprise-grade solution.