FARO RAG Pipeline - Proftaak Documentation

Student name: Sonny He
Student number: 5319587
Class: MA-NCA1
Semester coach: Erik Aerdts
Version: 1.0
Date: 2 Januari, 2025

1. Table of Contents

2. Introduction

2.1 Project Context

This document details my contributions to the FARO RAG (Retrieval-Augmented Generation) Pipeline proftaak project. FARO Automated Solutions needed a scalable cloud infrastructure to process documents and generate vector embeddings for intelligent search. The system had to handle document ingestion, text chunking, embedding generation, and semantic search across multiple storage backends.

2.2 System Overview

The FARO RAG Pipeline processes documents through a multi-stage workflow. When a document is uploaded, the Document Chunking Service splits it into smaller semantic chunks. These chunks get stored in S3, which triggers the Embeddings Engine Service to generate vector embeddings using AI models. The embeddings are then stored in both Qdrant for fast similarity search and PostgreSQL for long-term storage. Finally, the RAG Query Service retrieves relevant information and generates responses based on user queries.

The system runs on AWS infrastructure in the eu-central-1 region. The Document Chunking Service operates on port 8000 and handles text processing. The Embeddings Engine Service runs on port 8001 and generates vector representations of the text chunks. The RAG Query Service listens on port 8002 and handles user queries by searching through stored embeddings. All services run on an AWS EKS cluster with dual storage in Qdrant and PostgreSQL.

2.3 My Contributions

Within this proftaak, I was responsible for three main areas of the infrastructure.

First, I deployed and configured the EKS cluster infrastructure. This included setting up the node groups with appropriate scaling configurations and t3.medium instance types. I configured the IAM roles and OIDC provider that allow service accounts to securely access AWS resources. I also set up the network security and access controls for the cluster.

Second, I implemented the Embeddings Engine Service with a dual-storage architecture that writes to both Qdrant and PostgreSQL at the same time. I integrated the Portkey API that our teacher provided for embedding generation. This simplified API key management and gave us a unified interface for different embedding models. I built health check endpoints that monitor database connectivity and configured the service to handle batch processing triggered by S3 events from the chunking service.

Third, I set up the monitoring infrastructure using Prometheus and Grafana deployed via Helm charts on the EKS cluster. I created ServiceMonitors to collect metrics from both the Qdrant vector database and the RAG Query service. I built a custom dashboard to visualize RAG pipeline performance including vector counts, search latency, and result overlap between the two storage systems. I also configured a PostgreSQL exporter to monitor the RDS database.

Lastly, I configured the firewall rules using Terraform security groups to enforce a strict "Bastion-First" network strategy. I designed a layered security architecture where critical resources like the RDS database, Qdrant vector store and Kubernetes nodes reside in private subnets, completely isolated from the public internet. I implemented a VPN Bastion security group as the single secure entry point for administrative access to internal tools and dashboards. Additionally, I configured the public Load Balancer security group to strictly route web traffic to the cluster's ingress controller while blocking all other inbound requests, ensuring that the infrastructure adheres to the principle of least privilege.

3. EKS Cluster Infrastructure

3.1 Overview

The EKS cluster provides the foundation for running all RAG pipeline services. I deployed a managed Kubernetes cluster called `faro-rag-cluster` on AWS EKS version 1.34. The cluster runs in two private subnets across different availability zones for high availability. All application workloads run as containerized pods on worker nodes managed by EKS.

3.2 EKS Cluster Configuration

I configured the EKS cluster with specific settings for security and access control. The cluster uses API and ConfigMap authentication mode, which allows both modern EKS access entries and legacy aws-auth ConfigMap authentication. I enabled bootstrap permissions for the cluster creator to ensure initial admin access.

The cluster runs in private subnets to keep the control plane secure. I configured it to use two private subnets: `service_subnet` for running application pods and `db_private` for database connectivity. This network separation ensures that application traffic stays isolated from database traffic.

Code snippet from eks.tf:

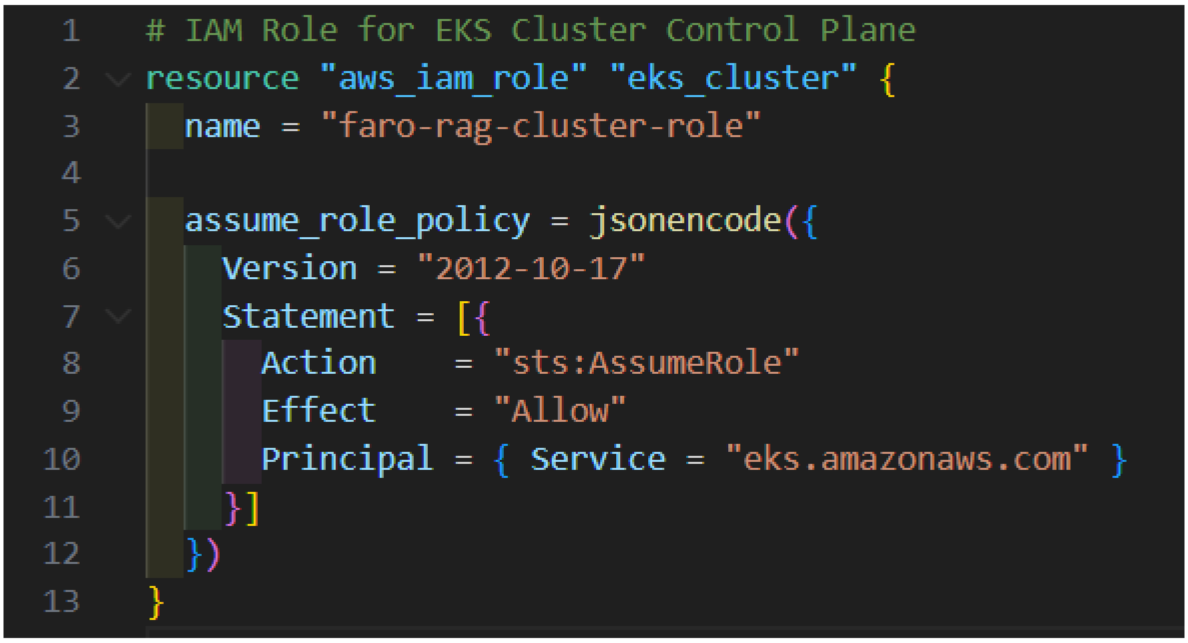
```
# The EKS Cluster Resource
resource "aws_eks_cluster" "main" {
  name     = "faro-rag-cluster"
  role_arn = aws_iam_role.eks_cluster.arn
  version  = "1.34"

  access_config {
    authentication_mode                         = "API_AND_CONFIG_MAP"
    bootstrap_cluster_creator_admin_permissions = true
  }

  vpc_config {
    # DEV SYNTAX: accessing subnets via map keys
    subnet_ids = [
      aws_subnet.private["service_subnet"].id,
      aws_subnet.private["db_private"].id
    ]
    security_group_ids = [aws_security_group.kuber.id]
  }

  depends_on = [
    aws_iam_role_policy_attachment.eks_cluster_policy
  ]
}
```

The cluster resource defines the control plane configuration. The `authentication_mode` setting allows both modern access entries (for SSO users) and ConfigMap-based access (for service accounts). The `bootstrap_cluster_creator_admin_permissions` ensures that whoever creates the cluster automatically gets admin access. The cluster spans two subnets for redundancy and uses a dedicated security group for network access control.

3.3 IAM Roles for EKS Cluster

The EKS control plane needs an IAM role to interact with other AWS services on your behalf. I created a dedicated IAM role that allows the EKS service to assume it and attached the required AWS-managed policy.

Code snippet from eks.tf:



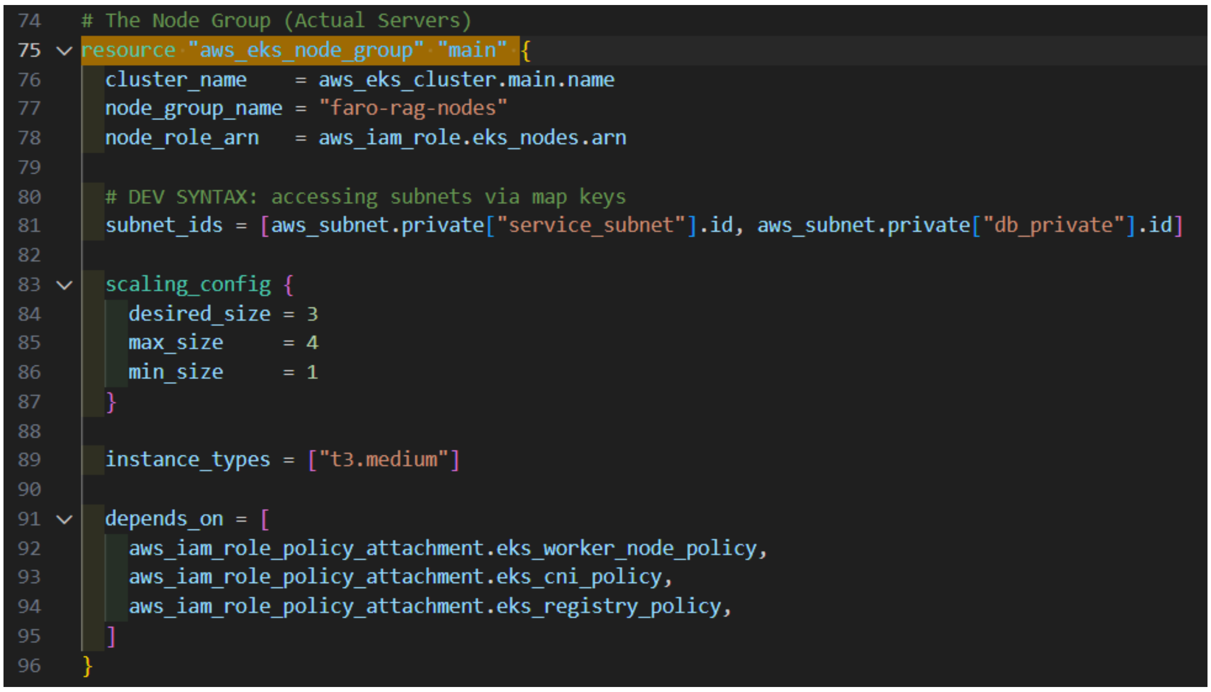
```
# IAM Role for EKS Cluster Control Plane
resource "aws_iam_role" "eks_cluster" {
  name = "faro-rag-cluster-role"

  assume_role_policy = jsonencode({
    Version = "2012-10-17"
    Statement = [{
      Action    = "sts:AssumeRole"
      Effect    = "Allow"
      Principal = { Service = "eks.amazonaws.com" }
    }]
  })
}
```

The assume role policy defines who can use this role. In this case, only the EKS service (`eks.amazonaws.com`) can assume it. The `AmazonEKSClusterPolicy` gives the control plane permissions to create and manage AWS resources like load balancers, security groups, and network interfaces that Kubernetes services need.

3.4 Node Group Configuration

The node group defines the EC2 instances that run your application pods. I configured a node group with t3.medium instances that can scale between 1 and 4 nodes based on workload demands. The nodes run in the same private subnets as the cluster for network consistency.

Code snippet from eks.tf:



```
# The Node Group (Actual Servers)
resource "aws_eks_node_group" "main" {
  cluster_name    = aws_eks_cluster.main.name
  node_group_name = "faro-rag-nodes"
  node_role_arn   = aws_iam_role.eks_nodes.arn

  # DEV SYNTAX: accessing subnets via map keys
  subnet_ids = [aws_subnet.private["service_subnet"].id, aws_subnet.private["db_private"].id]

  scaling_config {
    desired_size = 3
    max_size     = 4
    min_size     = 1
  }

  instance_types = ["t3.medium"]

  depends_on = [
    aws_iam_role_policy_attachment.eks_worker_node_policy,
    aws_iam_role_policy_attachment.eks_cni_policy,
    aws_iam_role_policy_attachment.eks_registry_policy,
  ]
}
```

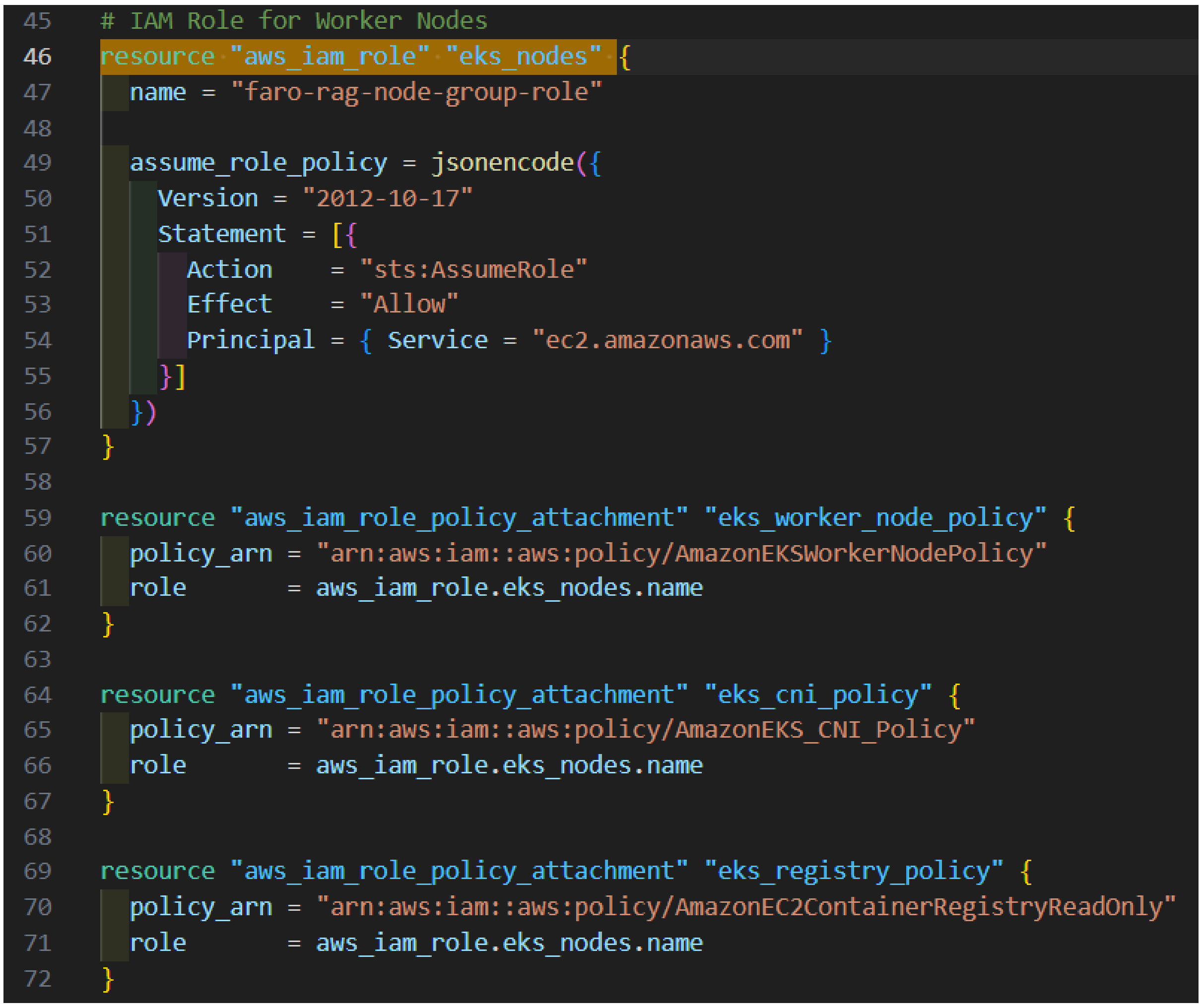
The node group starts with 3 nodes (desired_size) and can scale down to 1 or up to 4 nodes. I chose t3.medium instances because they provide a good balance of CPU and memory (2 vCPUs, 4GB RAM) for running multiple microservices. The `depends_on` ensures that all required IAM policies are attached before creating nodes.

Instance Type Rationale: t3.medium instances were chosen because each RAG service (chunking, embeddings, query) needs around 500MB-1GB of memory. With 4GB RAM per node, we can run 2-3 services per node with overhead for system processes and Kubernetes components.

3.5 IAM Roles for Worker Nodes

Worker nodes need different permissions than the control plane. They need to pull container images from ECR, connect to the cluster, and manage networking. I created a separate IAM role with three AWS-managed policies for these functions.

Code snippet from eks.tf:



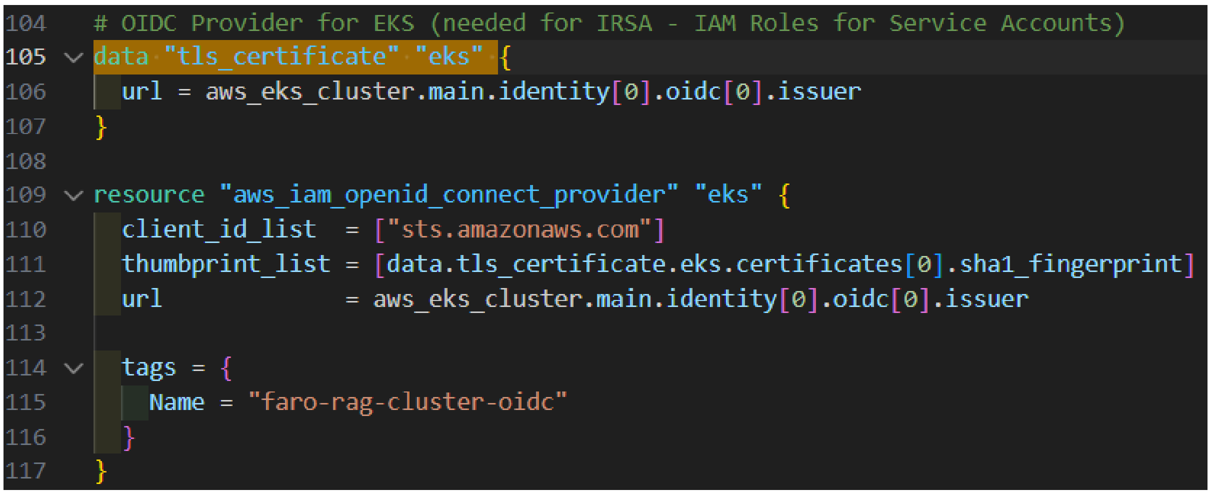
```hcl
# IAM Role for Worker Nodes
resource "aws_iam_role" "eks_nodes" {
  name = "faro-rag-node-group-role"

  assume_role_policy = jsonencode({
    Version = "2012-10-17"
    Statement = [{
      Action    = "sts:AssumeRole"
      Effect    = "Allow"
      Principal = { Service = "ec2.amazonaws.com" }
    }]
  })
}

resource "aws_iam_role_policy_attachment" "eks_worker_node_policy" {
  policy_arn = "arn:aws:iam::aws:policy/AmazonEKSWorkerNodePolicy"
  role       = aws_iam_role.eks_nodes.name
}

resource "aws_iam_role_policy_attachment" "eks_cni_policy" {
  policy_arn = "arn:aws:iam::aws:policy/AmazonEKS_CNI_Policy"
  role       = aws_iam_role.eks_nodes.name
}

resource "aws_iam_role_policy_attachment" "eks_registry_policy" {
  policy_arn = "arn:aws:iam::aws:policy/AmazonEC2ContainerRegistryReadOnly"
  role       = aws_iam_role.eks_nodes.name
}
```

The `AmazonEKSWorkerNodePolicy` allows nodes to connect to the EKS cluster and register themselves. The `AmazonEKS_CNI_Policy` enables the AWS VPC CNI plugin to assign IP addresses to pods from the VPC subnet. The `AmazonEC2ContainerRegistryReadOnly` policy lets nodes pull container images from ECR repositories.

3.6 OIDC Provider for IRSA

IRSA (IAM Roles for Service Accounts) allows pods to assume IAM roles without storing credentials. This is much more secure than embedding AWS access keys in pods. I set up an OIDC provider that establishes trust between the EKS cluster and AWS IAM.

Code snippet from eks.tf:



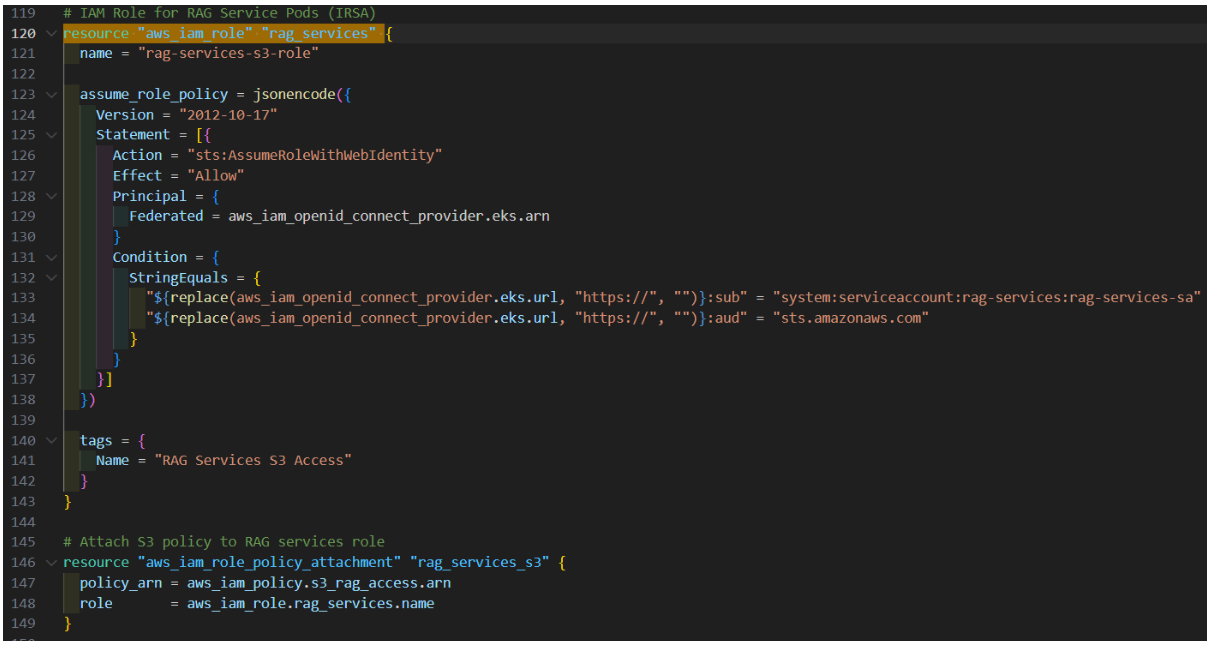
```
# OIDC Provider for EKS (needed for IRSA - IAM Roles for Service Accounts)
data "tls_certificate" "eks" {
  url = aws_eks_cluster.main.identity[0].oidc[0].issuer
}

resource "aws_iam_openid_connect_provider" "eks" {
  client_id_list  = ["sts.amazonaws.com"]
  thumbprint_list = [data.tls_certificate.eks.certificates[0].sha1_fingerprint]
  url             = aws_eks_cluster.main.identity[0].oidc[0].issuer

  tags = {
    Name = "faro-rag-cluster-oidc"
  }
}
```

The OIDC provider acts as a bridge between Kubernetes service accounts and IAM roles. When a pod needs AWS permissions, it uses its Kubernetes service account token to request temporary AWS credentials through the OIDC provider. This is more secure than static credentials because the tokens are short-lived and automatically rotated.

3.7 Service Account IAM Role

I created an IAM role specifically for the RAG services that need to access S3. This role uses IRSA to allow pods running under the `rag-services-sa` service account to assume it and access the S3 bucket.

Code snippet from eks.tf:



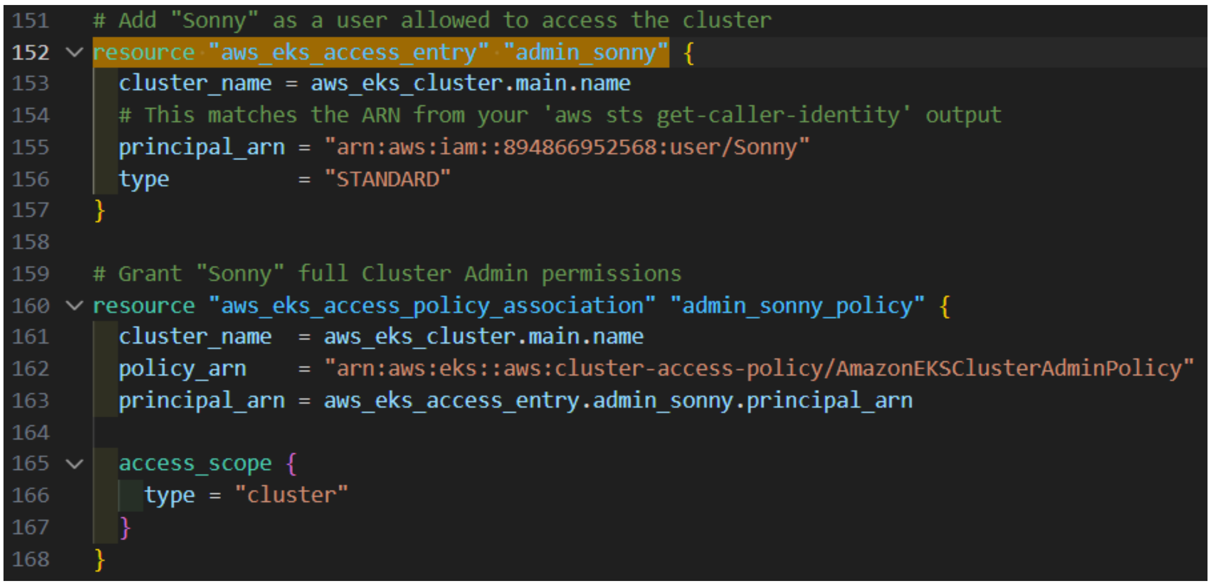
```
# IAM Role for RAG Service Pods (IRSA)
resource "aws_iam_role" "rag_services" {
  name = "rag-services-s3-role"

  assume_role_policy = jsonencode({
    Version = "2012-10-17"
    Statement = [{
      Action = "sts:AssumeRoleWithWebIdentity"
      Effect = "Allow"
      Principal = {
        Federated = aws_iam_openid_connect_provider.eks.arn
      }
      Condition = {
        StringEquals = {
          "${replace(aws_iam_openid_connect_provider.eks.url, "https://", "")}:sub" = "system:serviceaccount:rag-services:rag-services-sa"
          "${replace(aws_iam_openid_connect_provider.eks.url, "https://", "")}:aud" = "sts.amazonaws.com"
        }
      }
    }]
  })

  tags = {
    Name = "RAG Services S3 Access"
  }
}

# Attach S3 policy to RAG services role
resource "aws_iam_role_policy_attachment" "rag_services_s3" {
  policy_arn = aws_iam_policy.s3_rag_access.arn
  role       = aws_iam_role.rag_services.name
}
```

Explanation: The assume role policy has conditions that restrict which service account can use this role. Only the `rag-services-sa` service account in the `rag-services` namespace can assume it. This prevents other pods from accidentally or maliciously accessing S3. The S3 access policy is then attached to give the role permissions to read and write objects in the RAG documents bucket.

3.8 User Access Configuration

I configured user access to allow specific AWS users to manage the cluster through kubectl. I added an access entry for my user account and granted it cluster admin permissions.

Code snippet from eks.tf:



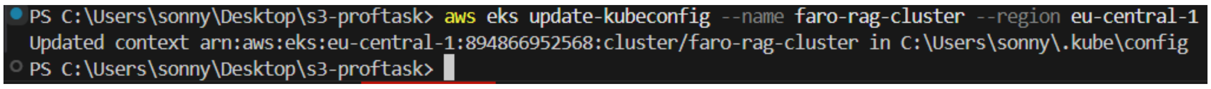
```
# Add "Sonny" as a user allowed to access the cluster
resource "aws_eks_access_entry" "admin_sonny" {
  cluster_name = aws_eks_cluster.main.name
  # This matches the ARN from your 'aws sts get-caller-identity' output
  principal_arn = "arn:aws:iam::894866952568:user/Sonny"
  type          = "STANDARD"
}

# Grant "Sonny" full Cluster Admin permissions
resource "aws_eks_access_policy_association" "admin_sonny_policy" {
  cluster_name  = aws_eks_cluster.main.name
  policy_arn    = "arn:aws:eks::aws:cluster-access-policy/AmazonEKSClusterAdminPolicy"
  principal_arn = aws_eks_access_entry.admin_sonny.principal_arn

  access_scope {
    type = "cluster"
  }
}
```

The access entry adds my IAM user to the cluster's access control list. The policy association grants cluster admin permissions, which allows full read and write access to all Kubernetes resources. This replaced the older aws-auth ConfigMap method and provides better auditability through CloudTrail.

3.9 Verification Commands

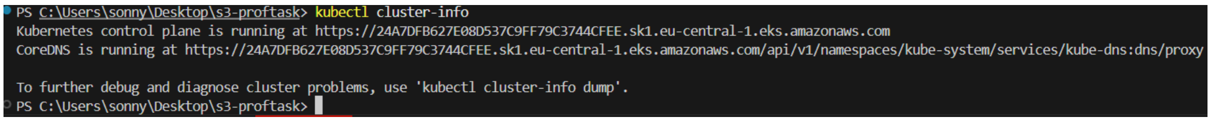
After deploying the infrastructure with Terraform, I verified the cluster was working correctly using these kubectl commands:

Update kubeconfig:



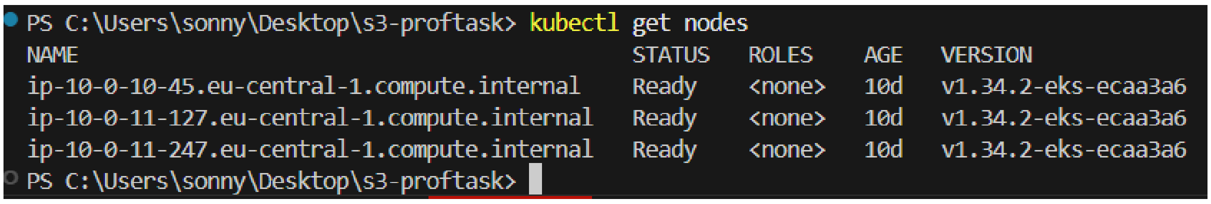
```
PS C:\Users\sonny\Desktop\s3-proftask> aws eks update-kubeconfig --name faro-rag-cluster --region eu-central-1
Updated context arn:aws:eks:eu-central-1:894866952568:cluster/faro-rag-cluster in C:\Users\sonny\.kube\config
PS C:\Users\sonny\Desktop\s3-proftask>
```

Check cluster connection:



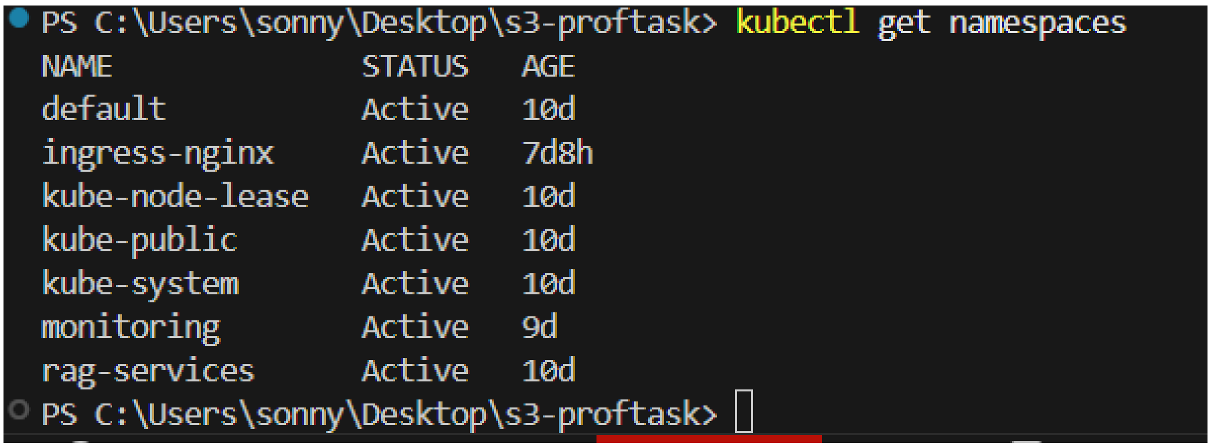
```
PS C:\Users\sonny\Desktop\s3-proftask> kubectl cluster-info
Kubernetes control plane is running at https://24A7DFB627E08D537C9FF79C3744CFEE.sk1.eu-central-1.eks.amazonaws.com
CoreDNS is running at https://24A7DFB627E08D537C9FF79C3744CFEE.sk1.eu-central-1.eks.amazonaws.com/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.
PS C:\Users\sonny\Desktop\s3-proftask>
```

Check node status:



```
PS C:\Users\sonny\Desktop\s3-proftask> kubectl get nodes
NAME                                         STATUS   ROLES    AGE   VERSION
ip-10-0-10-45.eu-central-1.compute.internal  Ready    <none>   10d   v1.34.2-eks-ecaa3a6
ip-10-0-11-127.eu-central-1.compute.internal Ready    <none>   10d   v1.34.2-eks-ecaa3a6
ip-10-0-11-247.eu-central-1.compute.internal Ready    <none>   10d   v1.34.2-eks-ecaa3a6
PS C:\Users\sonny\Desktop\s3-proftask>
```

Check namespaces:



```
PS C:\Users\sonny\Desktop\s3-proftask> kubectl get namespaces
NAME              STATUS   AGE
default           Active   10d
ingress-nginx     Active   7d8h
kube-node-lease   Active   10d
kube-public       Active   10d
kube-system       Active   10d
monitoring        Active   9d
rag-services      Active   10d
PS C:\Users\sonny\Desktop\s3-proftask>
```

This confirms that the cluster is operational and ready to deploy the RAG pipeline services.

4. Embeddings Engine Service

4.1 Overview

The Embeddings Engine Service is the core component responsible for converting text chunks into vector embeddings. When the Document Chunking Service saves chunks to S3, it triggers this service to process them. The service generates 1024-dimensional vectors using AI models and stores them in both Qdrant and PostgreSQL simultaneously. This dual-storage approach provides both fast similarity search capabilities and reliable long-term persistence.

The service exposes two main endpoints: `/embed` for generating single embeddings on demand and `/process/s3` for batch processing chunks from S3. It also includes a `/health` endpoint that reports the connection status to both database systems. The service runs as a Kubernetes deployment with 2 replicas for redundancy.

4.2 Dual-Storage Architecture

I implemented a dual-storage architecture that writes embeddings to both Qdrant and PostgreSQL at the same time. This design decision addresses different requirements that a single database couldn't satisfy.

Qdrant is optimized for vector similarity search. It uses specialized indexes (HNSW) that make finding similar vectors extremely fast, typically returning results in milliseconds. This makes it ideal for the RAG query service that needs to quickly find the most relevant chunks for a user's question. However, Qdrant runs on a single EC2 instance without built-in replication, which means data could be lost if that instance fails.

PostgreSQL with the pgvector extension provides reliable long-term storage with ACID guarantees. It supports backups, point-in-time recovery, and runs on AWS RDS with automated backups. If the Qdrant instance fails, we can rebuild it from PostgreSQL data. However, PostgreSQL's vector search is slower than Qdrant because it uses a general-purpose database engine that wasn't designed specifically for vector operations.

By using both systems, we get Qdrant's speed for queries and PostgreSQL's reliability for data durability. The tradeoff is increased complexity and storage costs, but for a production RAG system, this redundancy is worth it.

4.3 Portkey API Integration

I integrated the Portkey API for embedding generation. Our teacher provided us with a shared Portkey API key rather than having each student manage individual OpenAI or Cohere accounts. This approach offers several benefits for an educational setting.

Portkey acts as a unified gateway to multiple embedding model providers. Instead of implementing separate integrations for OpenAI, Cohere, and other providers, we make a single API call to Portkey which handles routing to the underlying model. This simplified our code and made it easy to switch between different embedding models without changing the application.

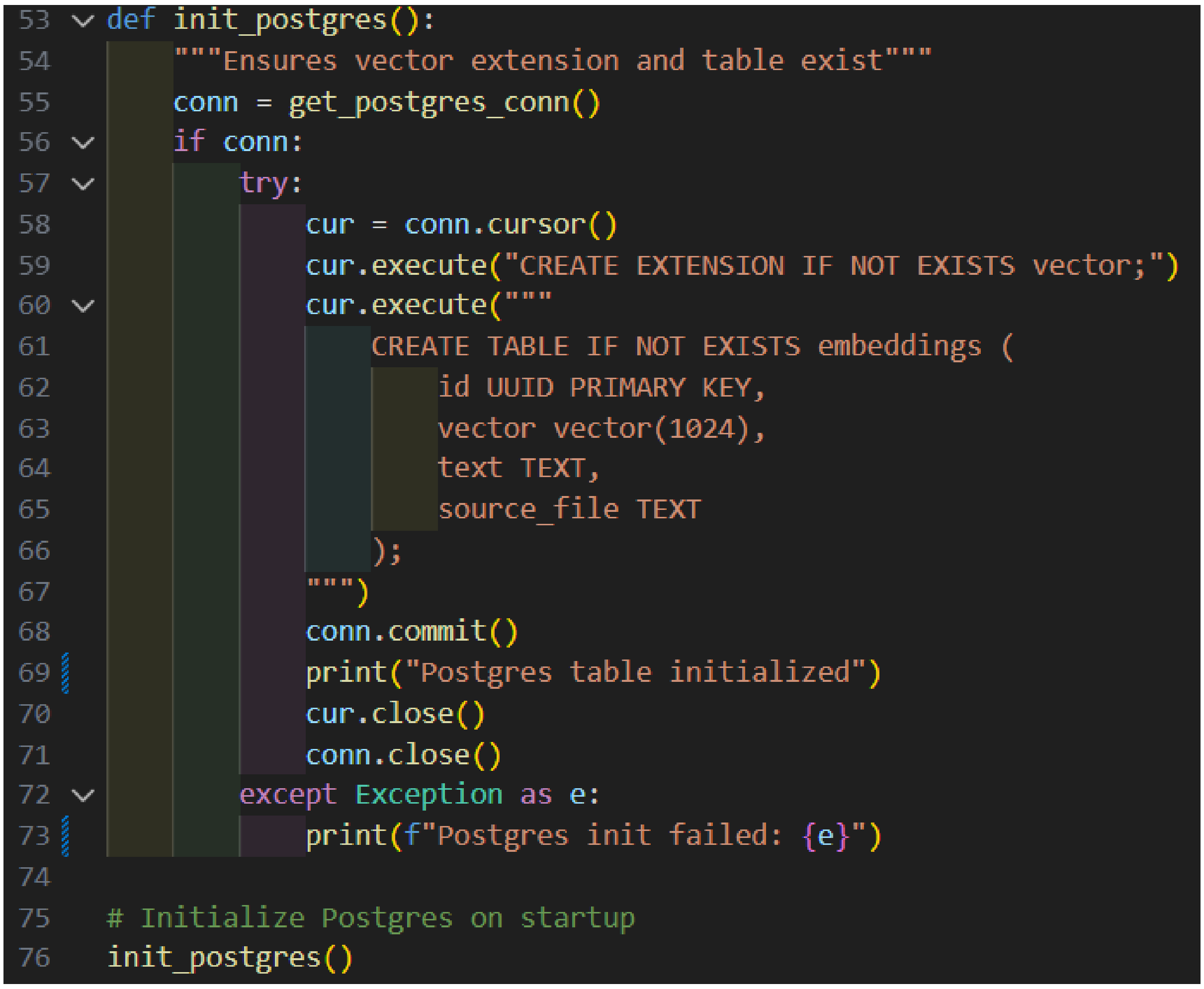
From a cost management perspective, Portkey allows the teacher to set spending limits and monitor usage across all students. This prevents accidental runaway costs that could happen if students had direct API access. Portkey also provides observability features that track how many embeddings are generated, latency statistics, and error rates, which helps with debugging issues.

The main downside is vendor lock-in to Portkey's API format, but for a learning environment where cost control and simplicity matter more than flexibility, this tradeoff makes sense.

4.4 PostgreSQL Initialization

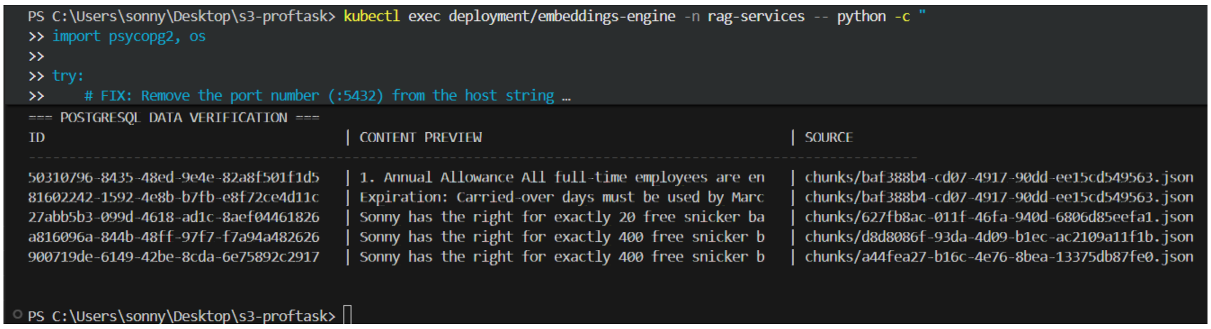
The service automatically initializes PostgreSQL when it starts up. It creates the vector extension if it doesn't exist and sets up the embeddings table with the correct schema.

Code snippet from main.py:



```python
def init_postgres():
    """Ensures vector extension and table exist"""
    conn = get_postgres_conn()
    if conn:
        try:
            cur = conn.cursor()
            cur.execute("CREATE EXTENSION IF NOT EXISTS vector;")
            cur.execute("""
                CREATE TABLE IF NOT EXISTS embeddings (
                    id UUID PRIMARY KEY,
                    vector vector(1024),
                    text TEXT,
                    source_file TEXT
                );
            """)
            conn.commit()
            print("Postgres table initialized")
            cur.close()
            conn.close()
        except Exception as e:
            print(f"Postgres init failed: {e}")

# Initialize Postgres on startup
init_postgres()
```

The `init_postgres()` function runs when the service starts. It first enables the `vector` extension which adds support for vector data types in PostgreSQL. Then it creates the embeddings table if it doesn't already exist. The `vector(1024)` type stores the 1024-dimensional embedding vectors. The function uses `IF NOT EXISTS` clauses so it's safe to run multiple times without errors. If the initialization fails (for example, if the database is unreachable), the service prints a warning but continues running so it can still use Qdrant.



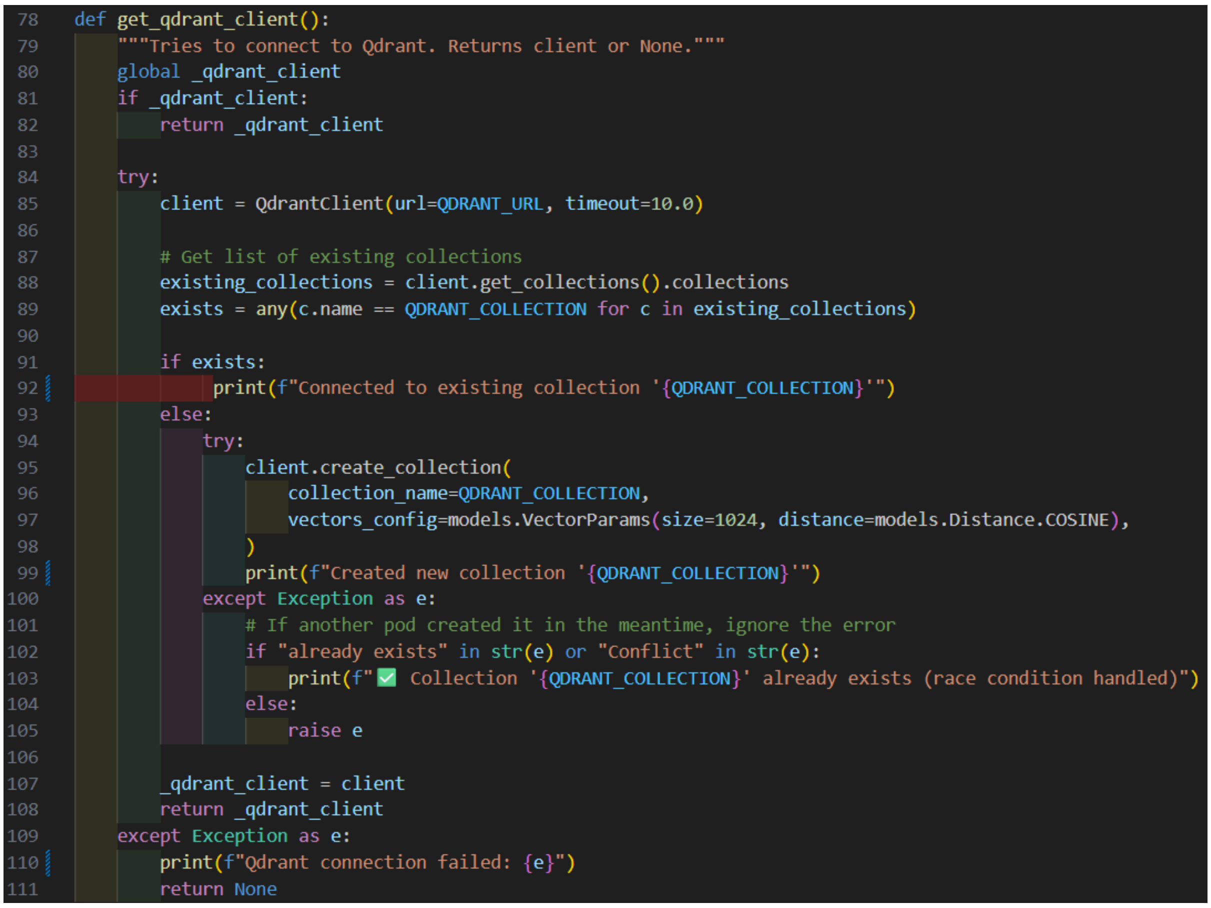
```
PS C:\Users\sonny\Desktop\s3-proftask> kubectl exec deployment/embeddings-engine -n rag-services -- python -c "
>> import psycopg2, os
>>
>> try:
>>     # FIX: Remove the port number (:5432) from the host string …
=== POSTGRESQL DATA VERIFICATION ===
ID                                   | CONTENT PREVIEW                          | SOURCE
----------------------------------------------------------------------------------------------------------------------
50310796-8435-48ed-9e4e-82a8f501f1d5 | 1. Annual Allowance All full-time employees are en | chunks/baf388b4-cd07-4917-90dd-ee15cd549563.json
81602242-1592-4e8b-b7fb-e8f72ce4d11c | Expiration: Carried-over days must be used by Marc | chunks/baf388b4-cd07-4917-90dd-ee15cd549563.json
27abb5b3-099d-4618-ad1c-8aef04461826 | Sonny has the right for exactly 20 free snicker ba | chunks/627fb8ac-011f-46fa-940d-6806d85eefa1.json
a816096a-844b-48ff-97f7-f7a94a482626 | Sonny has the right for exactly 400 free snicker b | chunks/d8d8086f-93da-4d09-b1ec-ac2109a11f1b.json
900719de-6149-42be-8cda-6e75892c2917 | Sonny has the right for exactly 400 free snicker b | chunks/a44fea27-b16c-4e76-8bea-13375db87fe0.json

PS C:\Users\sonny\Desktop\s3-proftask>
```

This shows the direct query to the RDS PostgreSQL embeddings table with vector IDs, source text and chunks.

4.5 Qdrant Connection and Collection Setup

The service connects to Qdrant lazily, meaning it only establishes the connection when needed. It also handles the creation of the vector collection if it doesn't exist.

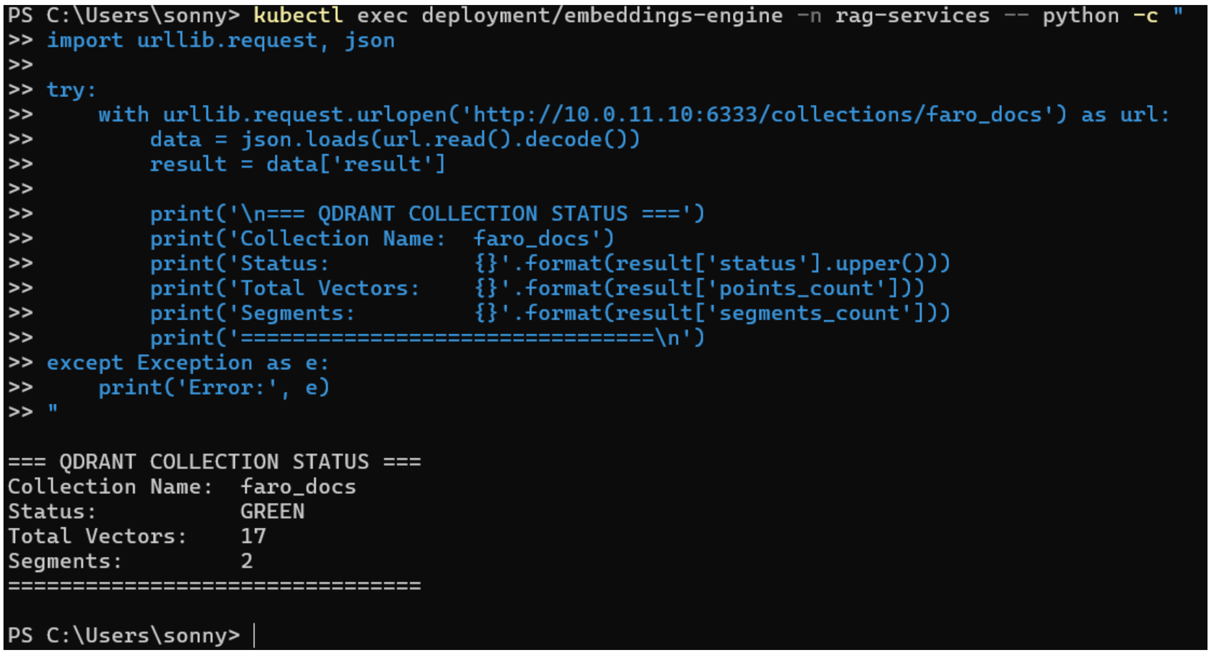
Code snippet from main.py:



```python
def get_qdrant_client():
    """Tries to connect to Qdrant. Returns client or None."""
    global _qdrant_client
    if _qdrant_client:
        return _qdrant_client

    try:
        client = QdrantClient(url=QDRANT_URL, timeout=10.0)

        # Get list of existing collections
        existing_collections = client.get_collections().collections
        exists = any(c.name == QDRANT_COLLECTION for c in existing_collections)

        if exists:
            print(f"Connected to existing collection '{QDRANT_COLLECTION}'")
        else:
            try:
                client.create_collection(
                    collection_name=QDRANT_COLLECTION,
                    vectors_config=models.VectorParams(size=1024, distance=models.Distance.COSINE),
                )
                print(f"Created new collection '{QDRANT_COLLECTION}'")
            except Exception as e:
                # If another pod created it in the meantime, ignore the error
                if "already exists" in str(e) or "Conflict" in str(e):
                    print(f"✅ Collection '{QDRANT_COLLECTION}' already exists (race condition handled)")
                else:
                    raise e

        _qdrant_client = client
        return _qdrant_client
    except Exception as e:
        print(f"Qdrant connection failed: {e}")
        return None
```

This function uses a global variable to cache the Qdrant client so we don't create new connections for every request. It first checks if the `faro_docs` collection exists. If not, it creates a collection configured for 1024-dimensional vectors using cosine distance for similarity calculations. Cosine distance works well for text embeddings because it measures the angle between vectors rather than absolute distance, which normalizes for vector magnitude.

The try-except block around collection creation handles a race condition where multiple pods might try to create the same collection simultaneously. If one pod creates it first, the others will get a "Conflict" error which we catch and ignore. If Qdrant is completely unreachable, the function returns `None` and the service continues using only PostgreSQL.



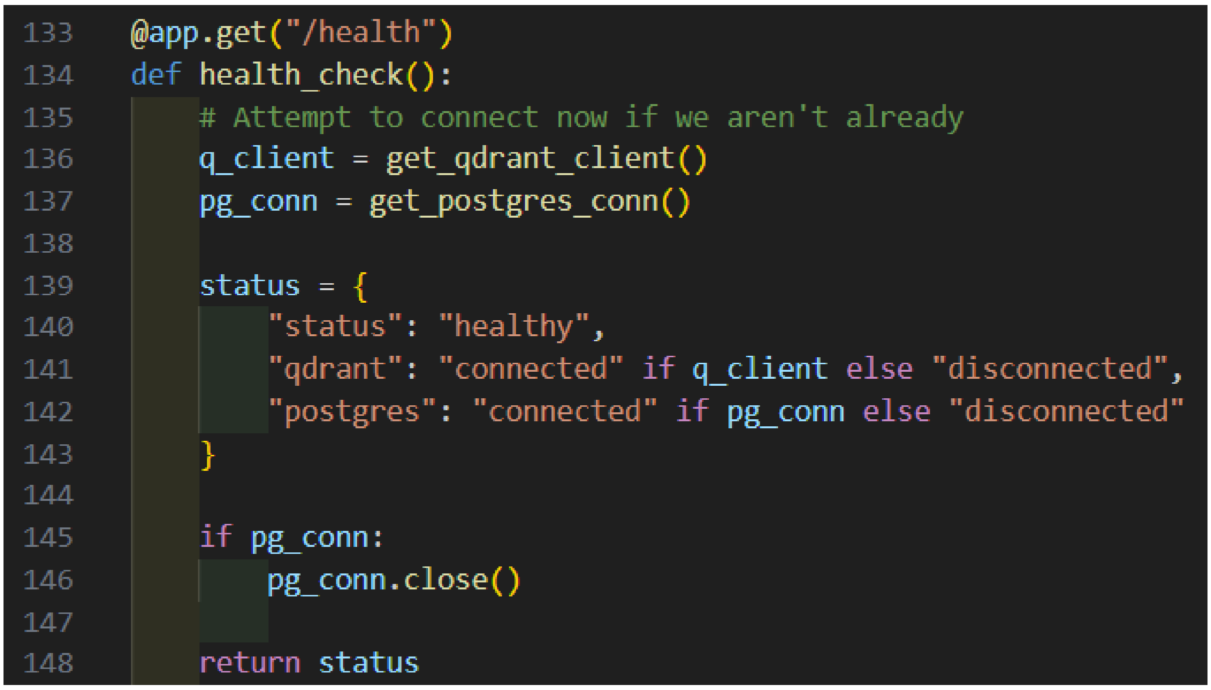
```
PS C:\Users\sonny> kubectl exec deployment/embeddings-engine -n rag-services -- python -c "
>> import urllib.request, json
>>
>> try:
>>     with urllib.request.urlopen('http://10.0.11.10:6333/collections/faro_docs') as url:
>>         data = json.loads(url.read().decode())
>>         result = data['result']
>>
>>         print('\n=== QDRANT COLLECTION STATUS ===')
>>         print('Collection Name:  faro_docs')
>>         print('Status:           {}'.format(result['status'].upper()))
>>         print('Total Vectors:    {}'.format(result['points_count']))
>>         print('Segments:         {}'.format(result['segments_count']))
>>         print('================================\n')
>> except Exception as e:
>>     print('Error:', e)
>> "

=== QDRANT COLLECTION STATUS ===
Collection Name:  faro_docs
Status:           GREEN
Total Vectors:    17
Segments:         2
================================

PS C:\Users\sonny>
```

This shows an internal API query to the qdrant-vector-db service showing the faro_docs collection is healthy (GREEN) and successfully indexing vectors. The Total Vectors count confirms that document chunks have been embedded and stored.

4.6 Health Check Endpoint

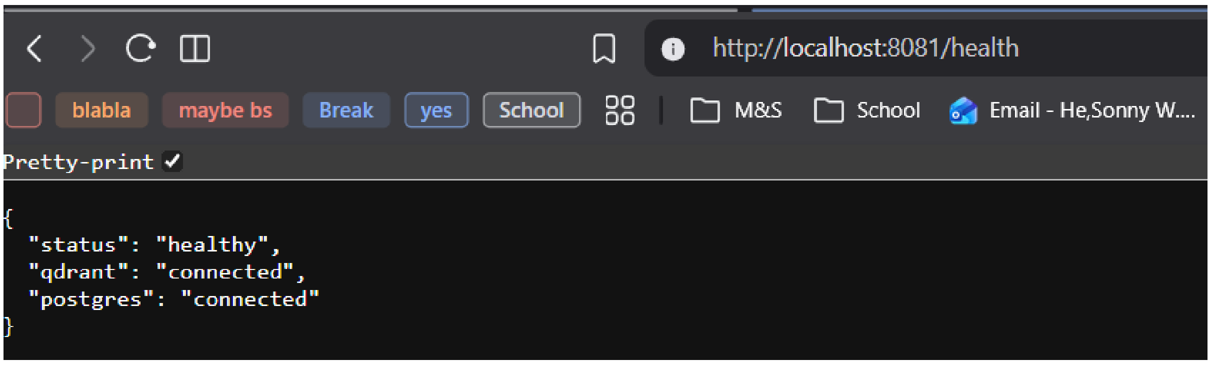
The health check endpoint reports the service status and database connectivity. This is used by Kubernetes liveness and readiness probes to determine if the pod is healthy.

Code snippet from main.py:



```python
@app.get("/health")
def health_check():
    # Attempt to connect now if we aren't already
    q_client = get_qdrant_client()
    pg_conn = get_postgres_conn()

    status = {
        "status": "healthy",
        "qdrant": "connected" if q_client else "disconnected",
        "postgres": "connected" if pg_conn else "disconnected"
    }

    if pg_conn:
        pg_conn.close()

    return status
```

The endpoint attempts to connect to both Qdrant and PostgreSQL, then returns their connection status. Even if one or both databases are down, the endpoint still returns HTTP 200 with "disconnected" status. This is intentional because the service can partially function with only one database available. Kubernetes readiness probes use this endpoint to determine if the pod should receive traffic.

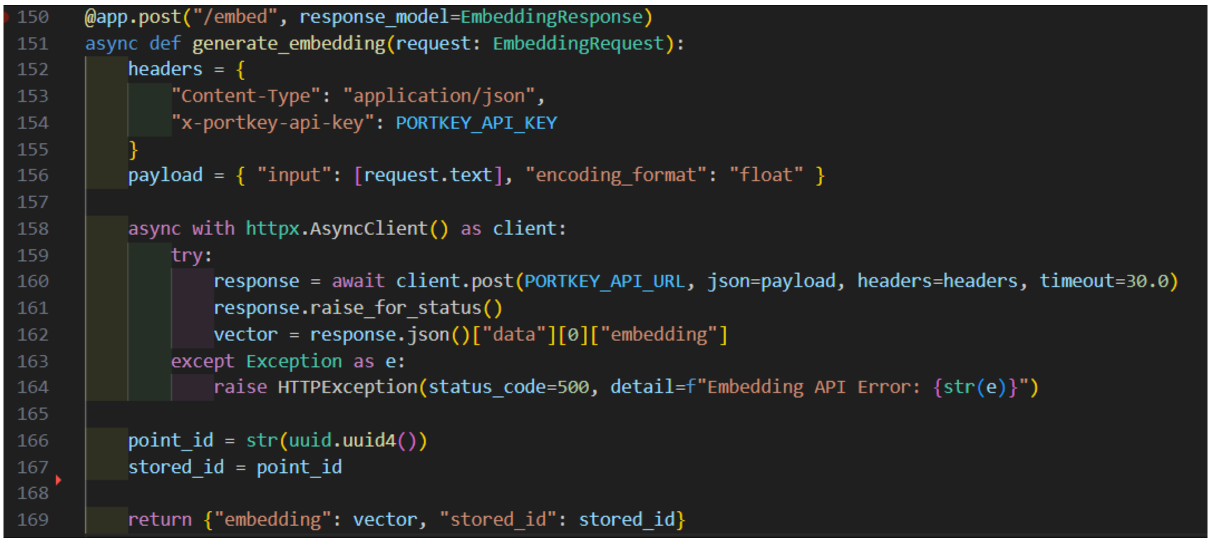


```
{
  "status": "healthy",
  "qdrant": "connected",
  "postgres": "connected"
}
```

4.7 Single Embedding Endpoint

The `/embed` endpoint generates a single embedding for a given text input without storing it. This is useful for testing or for applications that need to embed individual queries on-the-fly.

Code snippet from main.py:



```python
@app.post("/embed", response_model=EmbeddingResponse)
async def generate_embedding(request: EmbeddingRequest):
    headers = {
        "Content-Type": "application/json",
        "x-portkey-api-key": PORTKEY_API_KEY
    }
    payload = { "input": [request.text], "encoding_format": "float" }

    async with httpx.AsyncClient() as client:
        try:
            response = await client.post(PORTKEY_API_URL, json=payload, headers=headers, timeout=30.0)
            response.raise_for_status()
            vector = response.json()["data"][0]["embedding"]
        except Exception as e:
            raise HTTPException(status_code=500, detail=f"Embedding API Error: {str(e)}")

    point_id = str(uuid.uuid4())
    stored_id = point_id

    return {"embedding": vector, "stored_id": stored_id}
```

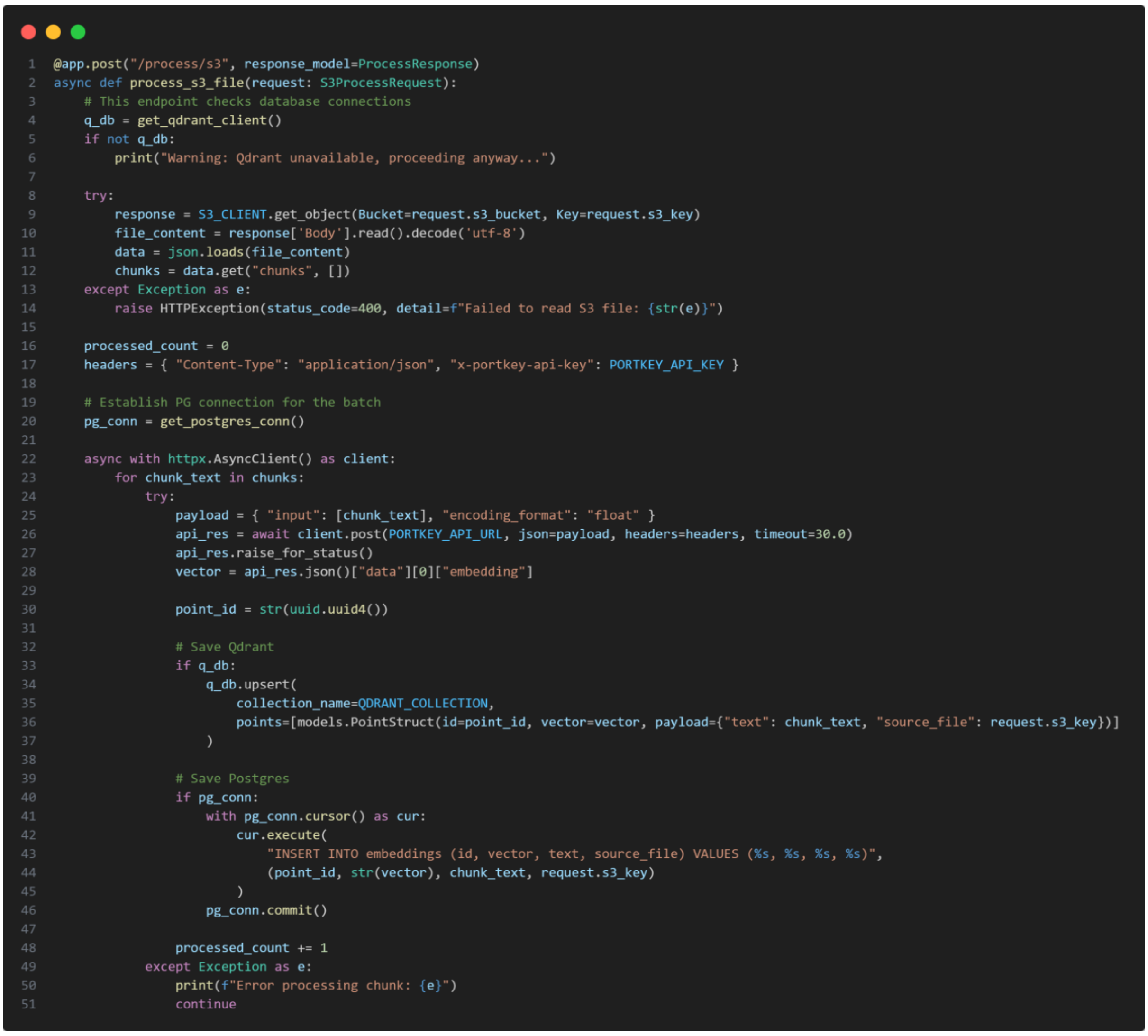
The endpoint calls the Portkey API provided by mr. Gleb to generate the embedding. This Portkey API endpoint gives access to the Cohere-embed-v3-multilingual embeddingsmodel hosted in Azure AI Foundry. We use an async HTTP client because embedding generation can take 1-2 seconds and we don't want to block the event loop. The Portkey API returns a 1024-dimensional float array which we extract from the response.

This endpoint does not store the embedding in any database. It only generates and returns the vector. The `stored_id` in the response is generated for consistency with the API contract but nothing is actually stored. This endpoint is primarily used by the RAG Query service when it needs to convert a user's search query into a vector for similarity search.

4.8 Batch Processing Endpoint

The `/process/s3` endpoint handles batch processing of chunks stored in S3. This is the main workflow trigger when the chunking service saves new documents.

Code snippet from main.py:



```python
@app.post("/process/s3", response_model=ProcessResponse)
async def process_s3_file(request: S3ProcessRequest):
    # This endpoint checks database connections
    q_db = get_qdrant_client()
    if not q_db:
        print("Warning: Qdrant unavailable, proceeding anyway...")

    try:
        response = S3_CLIENT.get_object(Bucket=request.s3_bucket, Key=request.s3_key)
        file_content = response['Body'].read().decode('utf-8')
        data = json.loads(file_content)
        chunks = data.get("chunks", [])
    except Exception as e:
        raise HTTPException(status_code=400, detail=f"Failed to read S3 file: {str(e)}")

    processed_count = 0
    headers = { "Content-Type": "application/json", "x-portkey-api-key": PORTKEY_API_KEY }

    # Establish PG connection for the batch
    pg_conn = get_postgres_conn()

    async with httpx.AsyncClient() as client:
        for chunk_text in chunks:
            try:
                payload = { "input": [chunk_text], "encoding_format": "float" }
                api_res = await client.post(PORTKEY_API_URL, json=payload, headers=headers, timeout=30.0)
                api_res.raise_for_status()
                vector = api_res.json()["data"][0]["embedding"]

                point_id = str(uuid.uuid4())

                # Save Qdrant
                if q_db:
                    q_db.upsert(
                        collection_name=QDRANT_COLLECTION,
                        points=[models.PointStruct(id=point_id, vector=vector, payload={"text": chunk_text, "source_file": request.s3_key})]
                    )

                # Save Postgres
                if pg_conn:
                    with pg_conn.cursor() as cur:
                        cur.execute(
                            "INSERT INTO embeddings (id, vector, text, source_file) VALUES (%s, %s, %s, %s)",
                            (point_id, str(vector), chunk_text, request.s3_key)
                        )
                    pg_conn.commit()

                processed_count += 1
            except Exception as e:
                print(f"Error processing chunk: {e}")
                continue
```

This endpoint first downloads the chunks file from S3 using the provided bucket and key. The file contains a JSON array of text chunks. It then processes each chunk sequentially, generating an embedding via the Portkey API and storing it in both databases.

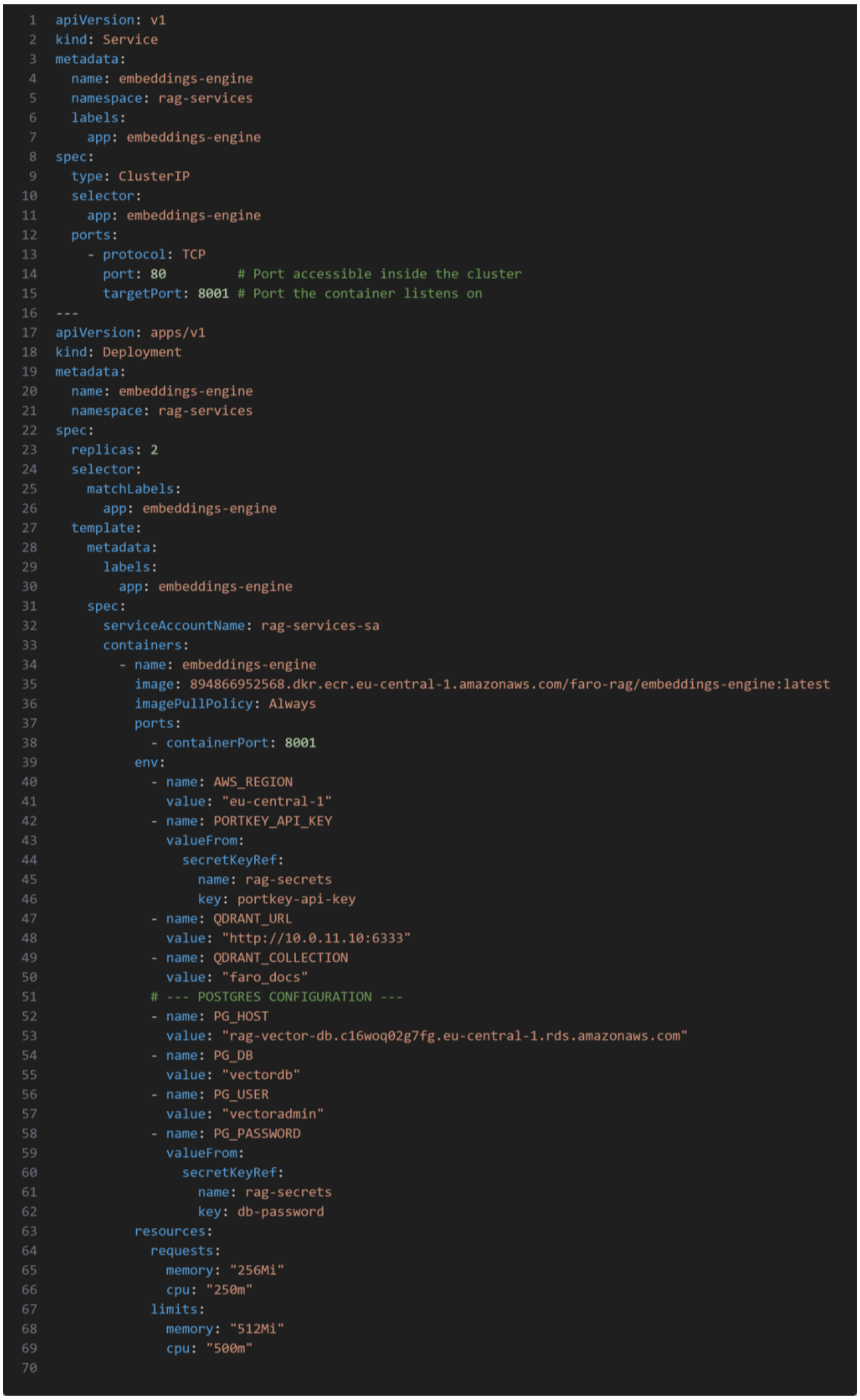
For each chunk, we generate a unique UUID that's used as the ID in both Qdrant and PostgreSQL. This ensures we can correlate records across both systems. We store the original chunk text and the source file path so we can trace embeddings back to their origin.

The loop uses error handling to skip individual chunk failures without stopping the entire batch. If one chunk fails to embed, we log the error and continue with the next chunk. This makes the process more resilient to transient API errors. The endpoint returns the total number of successfully processed chunks.

4.9 Kubernetes Deployment Configuration

The service runs as a Kubernetes deployment with specific resource limits and environment variable configuration.

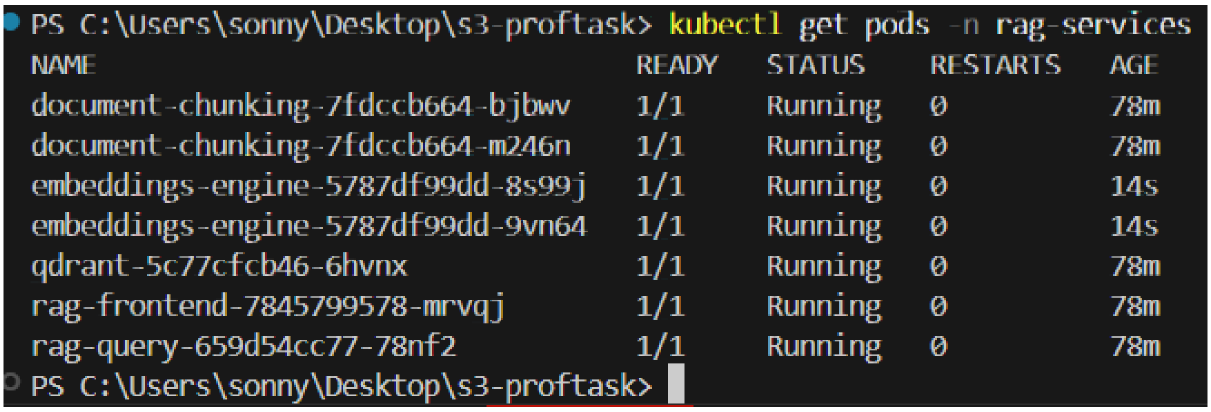
Code snippet from k8s/embeddings.yaml:



```yaml
apiVersion: v1
kind: Service
metadata:
  name: embeddings-engine
  namespace: rag-services
  labels:
    app: embeddings-engine
spec:
  type: ClusterIP
  selector:
    app: embeddings-engine
  ports:
    - protocol: TCP
      port: 80         # Port accessible inside the cluster
      targetPort: 8001 # Port the container listens on
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: embeddings-engine
  namespace: rag-services
spec:
  replicas: 2
  selector:
    matchLabels:
      app: embeddings-engine
  template:
    metadata:
      labels:
        app: embeddings-engine
    spec:
      serviceAccountName: rag-services-sa
      containers:
        - name: embeddings-engine
          image: 894866952568.dkr.ecr.eu-central-1.amazonaws.com/faro-rag/embeddings-engine:latest
          imagePullPolicy: Always
          ports:
            - containerPort: 8001
          env:
            - name: AWS_REGION
              value: "eu-central-1"
            - name: PORTKEY_API_KEY
              valueFrom:
                secretKeyRef:
                  name: rag-secrets
                  key: portkey-api-key
            - name: QDRANT_URL
              value: "http://10.0.11.10:6333"
            - name: QDRANT_COLLECTION
              value: "faro_docs"
            # --- POSTGRES CONFIGURATION ---
            - name: PG_HOST
              value: "rag-vector-db.c16woq02g7fg.eu-central-1.rds.amazonaws.com"
            - name: PG_DB
              value: "vectordb"
            - name: PG_USER
              value: "vectoradmin"
            - name: PG_PASSWORD
              valueFrom:
                secretKeyRef:
                  name: rag-secrets
                  key: db-password
          resources:
            requests:
              memory: "256Mi"
              cpu: "250m"
            limits:
              memory: "512Mi"
              cpu: "500m"
```

The deployment creates 2 replicas for redundancy. If one pod crashes, the other continues serving requests while Kubernetes restarts the failed pod. The `serviceAccountName` references the IRSA-enabled service account that allows pods to access S3.

Sensitive values like the Portkey API key and database password are stored in a Kubernetes Secret and injected as environment variables. This is more secure than hardcoding credentials in the image. The resource requests tell Kubernetes the minimum resources needed to run the pod, while limits prevent the pod from consuming too many resources and affecting other workloads.



```
PS C:\Users\sonny\Desktop\s3-proftask> kubectl get pods -n rag-services
NAME                                  READY   STATUS    RESTARTS   AGE
document-chunking-7fdccb664-bjbwv     1/1     Running   0          78m
document-chunking-7fdccb664-m246n     1/1     Running   0          78m
embeddings-engine-5787df99dd-8s99j    1/1     Running   0          14s
embeddings-engine-5787df99dd-9vn64    1/1     Running   0          14s
qdrant-5c77cfcb46-6hvnx               1/1     Running   0          78m
rag-frontend-7845799578-mrvqj         1/1     Running   0          78m
rag-query-659d54cc77-78nf2            1/1     Running   0          78m
PS C:\Users\sonny\Desktop\s3-proftask>
```

4.10 Environment Variables

The service uses environment variables for configuration to keep it flexible across different environments.

Variable	Value	Description
`AWS_REGION`	eu-central-1	AWS region for S3 access
`PORTKEY_API_KEY`	pk-xxx	API key for Portkey embedding service (from Secret)
`QDRANT_URL`	http://10.0.11.10:6333	URL of Qdrant vector database
`QDRANT_COLLECTION`	faro_docs	Name of the vector collection in Qdrant
`PG_HOST`	rag-vector-db.xxx.rds.amazonaws.com	PostgreSQL RDS endpoint
`PG_DB`	vectordb	PostgreSQL database name
`PG_USER`	vectoradmin	PostgreSQL username
`PG_PASSWORD`	xxx	PostgreSQL password (from Secret)

Using environment variables instead of hardcoded values makes the service portable. We can deploy the same container image to different environments (dev, staging, prod) just by changing the environment variables. Secrets are stored in Kubernetes rather than in the code or container image, which prevents credential leakage if the image is accidentally published.

5. Monitoring Implementation

5.1 Overview

I implemented a complete monitoring stack for the RAG pipeline using Prometheus and Grafana deployed via Helm charts on the EKS cluster. The monitoring system collects metrics from all components including the Qdrant vector database, PostgreSQL RDS instance, and the RAG services themselves. I built a custom Grafana dashboard specifically designed to track RAG pipeline performance metrics like vector counts, search latency, and result accuracy.

The monitoring infrastructure runs in its own monitoring namespace to isolate it from application workloads. Prometheus scrapes metrics from various targets every 15 seconds and stores them for 30 days. Grafana provides visualization and is accessible via an internal AWS load balancer so it can only be reached through the VPN connection.

5.2 Prometheus and Grafana Helm Deployment

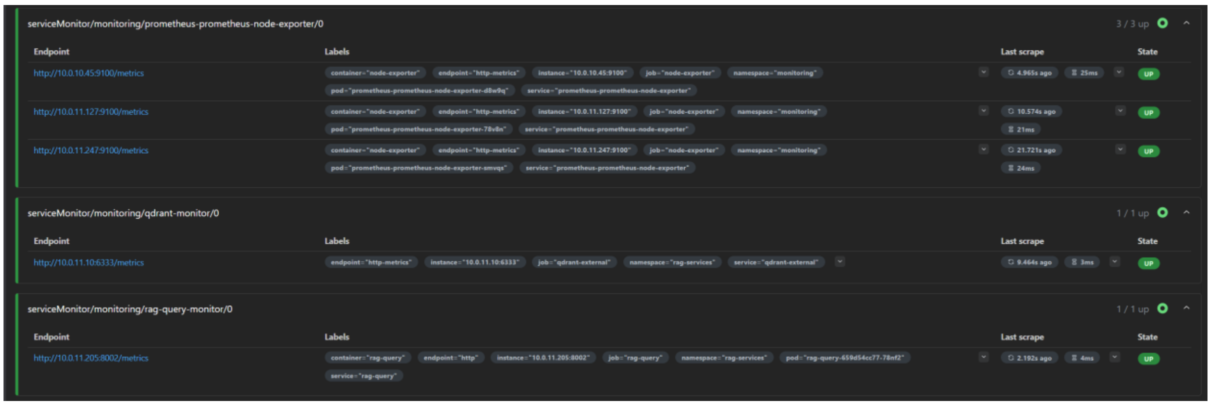
I deployed the Prometheus and Grafana stack using the community-maintained kube-prometheus-stack Helm chart. This chart includes Prometheus, Grafana, Alertmanager, and several pre-configured dashboards for Kubernetes monitoring.

Code snippet from monitoring.tf:

```
resource "helm_release" "kube_prometheus_stack" {
  name             = "prometheus"
  repository       = "https://prometheus-community.github.io/helm-charts"
  chart            = "kube-prometheus-stack"
  namespace        = "monitoring"
  create_namespace = true

  set {
    name  = "grafana.adminPassword"
    value = "admin123"
  }

  # Allow Prometheus to scrape services across all namespaces
  set {
    name  = "prometheus.prometheusSpec.serviceMonitorSelectorNilUsesHelmValues"
    value = "false"
  }

  # --- VPN ACCESS CONFIGURATION ---
  # This gives Grafana an internal IP (e.g., 10.0.x.x) reachable via VPN
  set {
    name  = "grafana.service.type"
    value = "LoadBalancer"
  }

  set {
    name  = "grafana.service.annotations.service\\.beta\\.kubernetes\\.io/aws-load-balancer-internal"
    value = "true"
    type  = "string"
  }
}
```

The Helm chart deployment creates the monitoring namespace automatically if it doesn't exist. I set a simple admin password for Grafana access during development. The serviceMonitorSelectorNilUsesHelmValues setting is critical because it allows Prometheus to discover ServiceMonitor resources across all namespaces, not just the monitoring namespace. Without this, Prometheus would ignore the ServiceMonitors I created for the RAG services.

The Grafana service is configured as a LoadBalancer with the aws-load-balancer-internal annotation. This creates an internal AWS Network Load Balancer that only has a private IP address within the VPC. This means Grafana is only accessible through the VPN connection, which prevents unauthorized external access to monitoring data.



serviceMonitor/monitoring/prometheus-prometheus-node-exporter/0 — 3 / 3 up

Endpoint	Labels	Last scrape	State
http://10.0.10.45:9100/metrics	container="node-exporter" endpoint="http-metrics" instance="10.0.10.45:9100" job="node-exporter" namespace="monitoring" pod="prometheus-prometheus-node-exporter-d8w9q" service="prometheus-prometheus-node-exporter"	4.965s ago 25ms	UP
http://10.0.11.127:9100/metrics	container="node-exporter" endpoint="http-metrics" instance="10.0.11.127:9100" job="node-exporter" namespace="monitoring" pod="prometheus-prometheus-node-exporter-78v8n" service="prometheus-prometheus-node-exporter"	10.574s ago 21ms	UP
http://10.0.11.247:9100/metrics	container="node-exporter" endpoint="http-metrics" instance="10.0.11.247:9100" job="node-exporter" namespace="monitoring" pod="prometheus-prometheus-node-exporter-smvqs" service="prometheus-prometheus-node-exporter"	21.721s ago 24ms	UP

serviceMonitor/monitoring/qdrant-monitor/0 — 1 / 1 up

Endpoint	Labels	Last scrape	State
http://10.0.11.10:6333/metrics	endpoint="http-metrics" instance="10.0.11.10:6333" job="qdrant-external" namespace="rag-services" service="qdrant-external"	9.464s ago 3ms	UP

serviceMonitor/monitoring/rag-query-monitor/0 — 1 / 1 up

Endpoint	Labels	Last scrape	State
http://10.0.11.205:8002/metrics	container="rag-query" endpoint="http" instance="10.0.11.205:8002" job="rag-query" namespace="rag-services" pod="rag-query-659d54cc77-78nf2" service="rag-query"	2.192s ago 4ms	UP

The Prometheus Service Discovery page (/targets) showing the rag-query, qdrant-db, and postgres-exporter endpoints are successfully discovered and in the **UP** state, confirming that metric scraping is active for all pipeline components.

5.3 PostgreSQL Exporter Configuration

I deployed a PostgreSQL exporter to collect database metrics from the RDS instance. The exporter connects to PostgreSQL and exposes metrics like connection count, query performance, and table statistics in Prometheus format.

Code snippet from monitoring.tf:

```
# Postgres Exporter (To monitor RDS)
resource "helm_release" "postgres_exporter" {
  name       = "postgres-exporter"
  repository = "https://prometheus-community.github.io/helm-charts"
  chart      = "prometheus-postgres-exporter"
  namespace  = "monitoring"

  depends_on = [helm_release.kube_prometheus_stack]

  set {
    name  = "config.datasource.host"
    value = aws_db_instance.vector_db.address
  }

  set {
    name  = "config.datasource.user"
    value = var.db_username
  }

  set {
    name  = "config.datasource.password"
    value = var.db_password
  }

  set {
    name  = "config.datasource.sslmode"
    value = "require"
  }

  set {
    name  = "config.datasource.database"
    value = "vectordb"
  }

  # Enable the ServiceMonitor so Prometheus finds it automatically
  set {
    name  = "serviceMonitor.enabled"
    value = "true"
  }
}

```

The PostgreSQL exporter is deployed as a separate Helm release that depends on the Prometheus stack being deployed first. I configure it with the RDS endpoint and credentials so it can connect to the database. The sslmode is set to "require" because AWS RDS enforces SSL connections for security.

The serviceMonitor.enabled setting tells the Helm chart to create a ServiceMonitor resource automatically. This ServiceMonitor tells Prometheus where to scrape PostgreSQL metrics from. Without this, Prometheus wouldn't know about the exporter. The exporter queries PostgreSQL system tables to collect metrics like active connections, transaction rates, and table sizes.

5.4 Qdrant ServiceMonitor Configuration

Qdrant runs on an EC2 instance outside the Kubernetes cluster, which requires a special configuration to make it visible to Prometheus. I created a Kubernetes Service and Endpoints resource that bridge the external Qdrant instance into the cluster's service discovery.

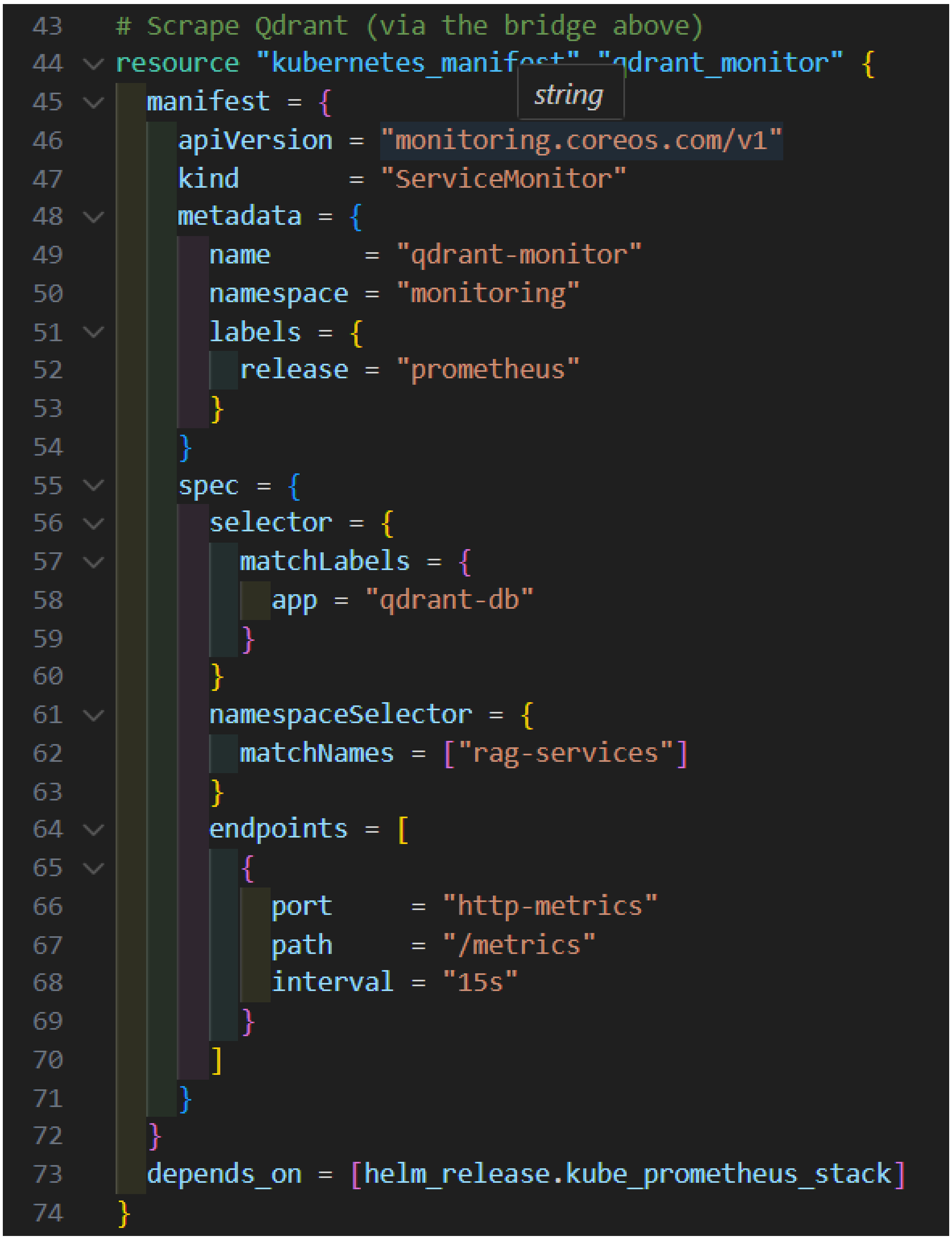
Code snippet from monitoring-targets.tf:

```
# Define a Service without a selector (Headless-ish)
resource "kubernetes_service" "qdrant_external" {
  metadata {
    name      = "qdrant-external"
    namespace = "rag-services"
    labels = {
      app = "qdrant-db"
    }
  }
  spec {
    port {
      name        = "http-metrics"
      port        = 6333
      target_port = 6333
    }
  }
}

# Manually map the Service to the EC2 IP
resource "kubernetes_endpoints" "qdrant_external" {
  metadata {
    name      = "qdrant-external"
    namespace = "rag-services"
  }
  subset {
    address {
      ip = "10.0.11.10" # Static IP from qdrant.tf
    }
    port {
      name     = "http-metrics"
      port     = 6333
    }
  }
}
```

Normally, Kubernetes Services automatically create Endpoints based on pod selectors. But Qdrant doesn't run as a pod, so I created a Service without a selector. Then I manually created an Endpoints resource that points to Qdrant's EC2 private IP address (10.0.11.10). This tricks Kubernetes into thinking Qdrant is a normal service that Prometheus can scrape.

The Service and Endpoints both use port 6333, which is Qdrant's default HTTP port. Qdrant exposes Prometheus metrics on the /metrics endpoint at this port. This bridging pattern is useful whenever you need to integrate external services into Kubernetes service discovery.

ServiceMonitor for Qdrant:



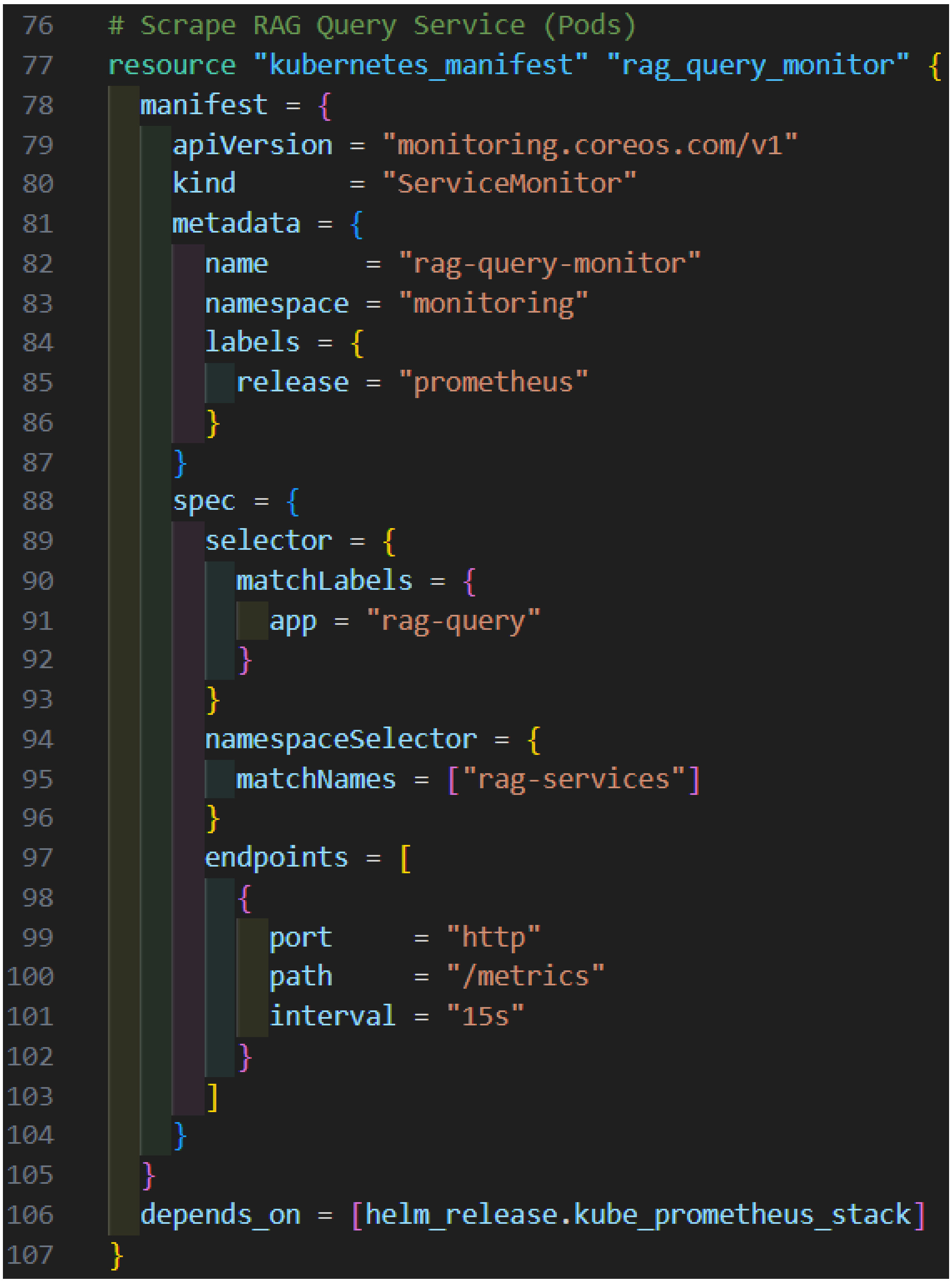
```
# Scrape Qdrant (via the bridge above)
resource "kubernetes_manifest" "qdrant_monitor" {
  manifest = {
    apiVersion = "monitoring.coreos.com/v1"
    kind       = "ServiceMonitor"
    metadata = {
      name      = "qdrant-monitor"
      namespace = "monitoring"
      labels = {
        release = "prometheus"
      }
    }
    spec = {
      selector = {
        matchLabels = {
          app = "qdrant-db"
        }
      }
      namespaceSelector = {
        matchNames = ["rag-services"]
      }
      endpoints = [
        {
          port     = "http-metrics"
          path     = "/metrics"
          interval = "15s"
        }
      ]
    }
  }
  depends_on = [helm_release.kube_prometheus_stack]
}
```

The ServiceMonitor tells Prometheus to look for services with the label app=qdrant-db in the rag-services namespace. It scrapes the /metrics endpoint every 15 seconds. The release=prometheus label is important because the Prometheus Operator uses this label to discover which ServiceMonitors belong to this Prometheus instance. Without it, the ServiceMonitor would be ignored.

5.5 RAG Query Service ServiceMonitor

I also created a ServiceMonitor for the RAG Query service to track query performance and latency metrics that the service exposes.

Code snippet from monitoring-targets.tf:



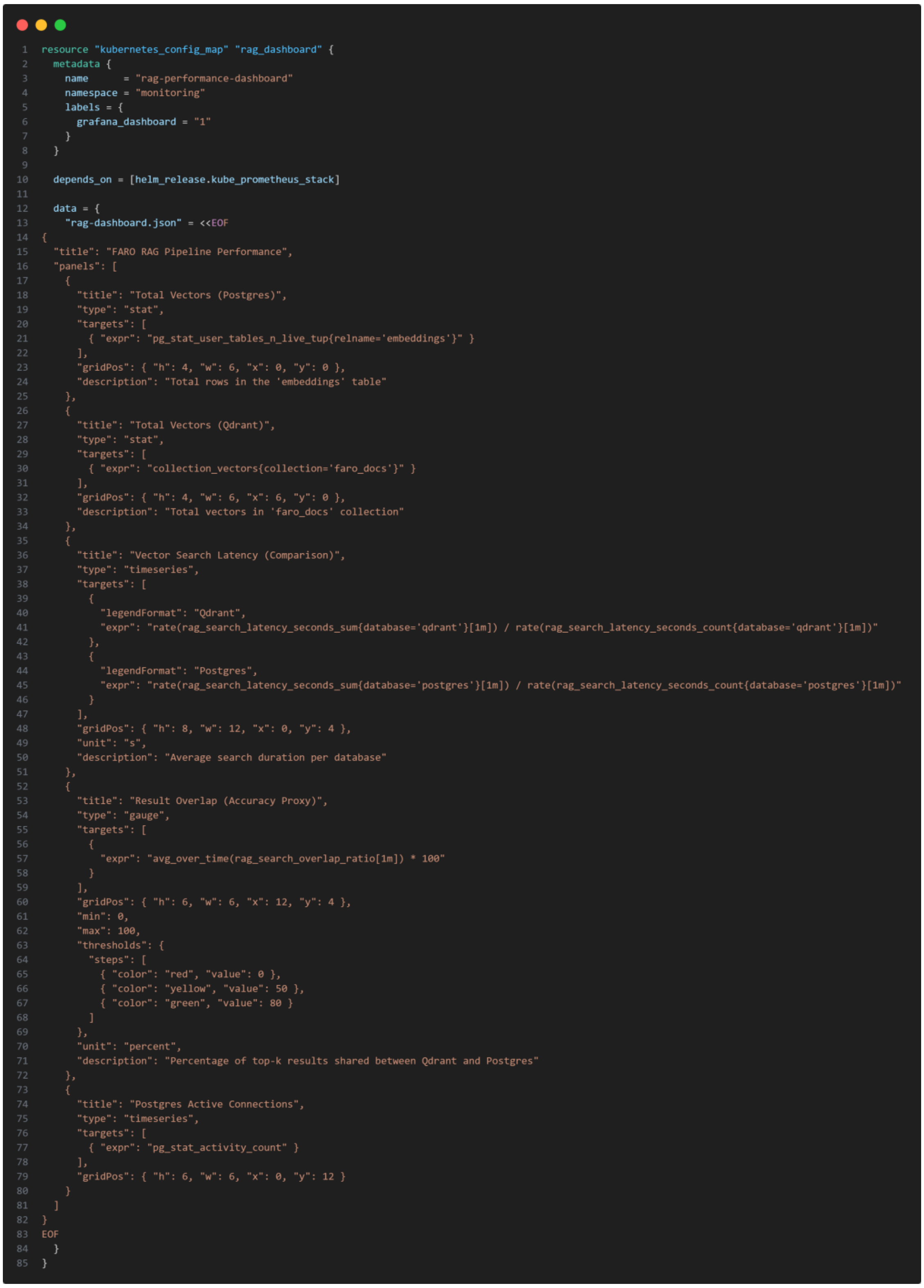
```
# Scrape RAG Query Service (Pods)
resource "kubernetes_manifest" "rag_query_monitor" {
  manifest = {
    apiVersion = "monitoring.coreos.com/v1"
    kind       = "ServiceMonitor"
    metadata = {
      name      = "rag-query-monitor"
      namespace = "monitoring"
      labels = {
        release = "prometheus"
      }
    }
    spec = {
      selector = {
        matchLabels = {
          app = "rag-query"
        }
      }
      namespaceSelector = {
        matchNames = ["rag-services"]
      }
      endpoints = [
        {
          port     = "http"
          path     = "/metrics"
          interval = "15s"
        }
      ]
    }
  }
  depends_on = [helm_release.kube_prometheus_stack]
}
```

This ServiceMonitor works the same way as the Qdrant monitor but targets the RAG Query service pods. The RAG Query service uses the Prometheus FastAPI Instrumentator library which automatically exposes standard metrics like request count, request duration, and HTTP status codes. It also exposes custom metrics for RAG-specific operations like search latency and result overlap that I defined in the service code.

5.6 Custom RAG Performance Dashboard

I built a custom Grafana dashboard specifically for monitoring RAG pipeline performance. The dashboard includes panels for vector counts, search latency comparison, result accuracy, and database connections.

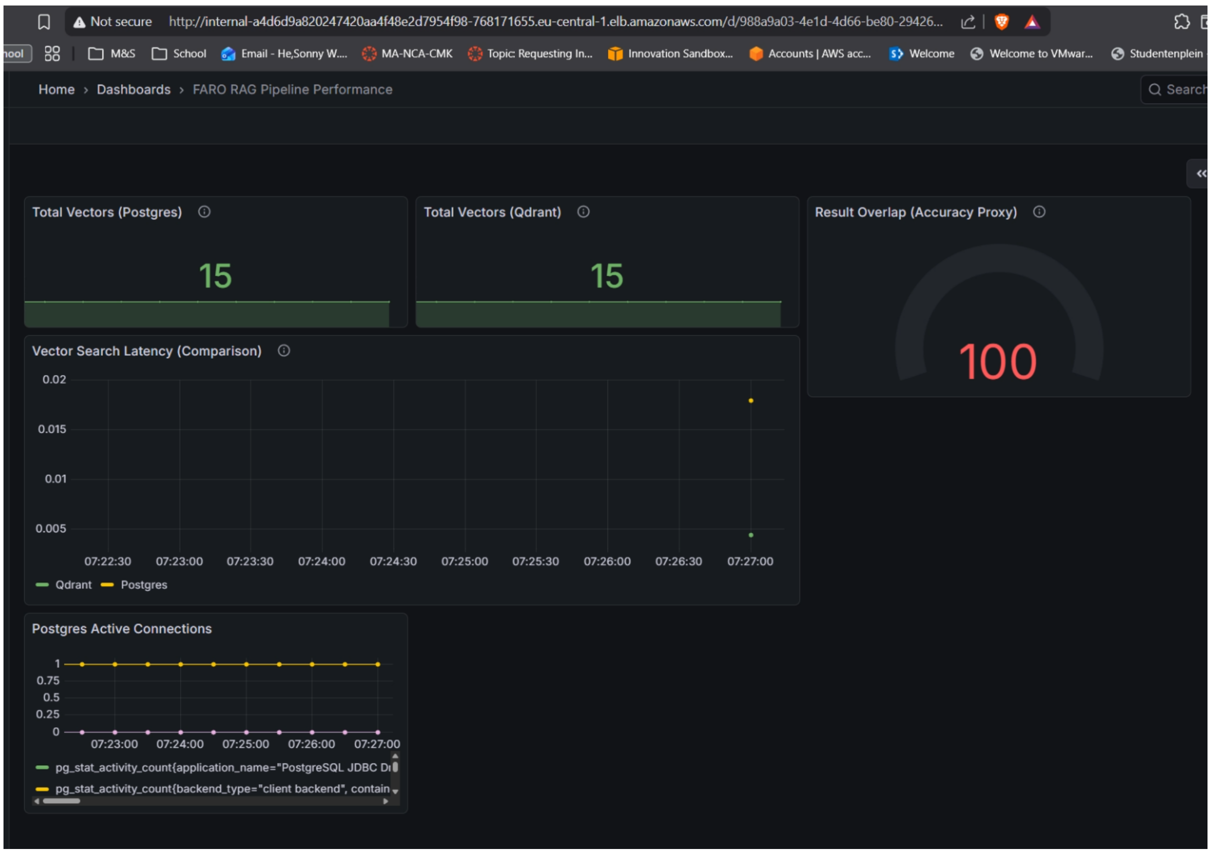
Code snippet from dashboard.tf:



```
resource "kubernetes_config_map" "rag_dashboard" {
  metadata {
    name      = "rag-performance-dashboard"
    namespace = "monitoring"
    labels = {
      grafana_dashboard = "1"
    }
  }

  depends_on = [helm_release.kube_prometheus_stack]

  data = {
    "rag-dashboard.json" = <<EOF
{
  "title": "FARO RAG Pipeline Performance",
  "panels": [
    {
      "title": "Total Vectors (Postgres)",
      "type": "stat",
      "targets": [
        { "expr": "pg_stat_user_tables_n_live_tup{relname='embeddings'}" }
      ],
      "gridPos": { "h": 4, "w": 6, "x": 0, "y": 0 },
      "description": "Total rows in the 'embeddings' table"
    },
    {
      "title": "Total Vectors (Qdrant)",
      "type": "stat",
      "targets": [
        { "expr": "collection_vectors{collection='faro_docs'}" }
      ],
      "gridPos": { "h": 4, "w": 6, "x": 6, "y": 0 },
      "description": "Total vectors in 'faro_docs' collection"
    },
    {
      "title": "Vector Search Latency (Comparison)",
      "type": "timeseries",
      "targets": [
        {
          "legendFormat": "Qdrant",
          "expr": "rate(rag_search_latency_seconds_sum{database='qdrant'}[1m]) / rate(rag_search_latency_seconds_count{database='qdrant'}[1m])"
        },
        {
          "legendFormat": "Postgres",
          "expr": "rate(rag_search_latency_seconds_sum{database='postgres'}[1m]) / rate(rag_search_latency_seconds_count{database='postgres'}[1m])"
        }
      ],
      "gridPos": { "h": 8, "w": 12, "x": 0, "y": 4 },
      "unit": "s",
      "description": "Average search duration per database"
    },
    {
      "title": "Result Overlap (Accuracy Proxy)",
      "type": "gauge",
      "targets": [
        {
          "expr": "avg_over_time(rag_search_overlap_ratio[1m]) * 100"
        }
      ],
      "gridPos": { "h": 6, "w": 6, "x": 12, "y": 4 },
      "min": 0,
      "max": 100,
      "thresholds": {
        "steps": [
          { "color": "red", "value": 0 },
          { "color": "yellow", "value": 50 },
          { "color": "green", "value": 80 }
        ]
      },
      "unit": "percent",
      "description": "Percentage of top-k results shared between Qdrant and Postgres"
    },
    {
      "title": "Postgres Active Connections",
      "type": "timeseries",
      "targets": [
        { "expr": "pg_stat_activity_count" }
      ],
      "gridPos": { "h": 6, "w": 6, "x": 0, "y": 12 }
    }
  ]
}
EOF
  }
}
```

The dashboard is stored as a Kubernetes ConfigMap with the label grafana_dashboard=1. Grafana watches for ConfigMaps with this label and automatically imports them as dashboards. This approach is better than manually creating dashboards because it's version controlled and can be recreated automatically if Grafana is redeployed.

5.7 Dashboard Panels Explained

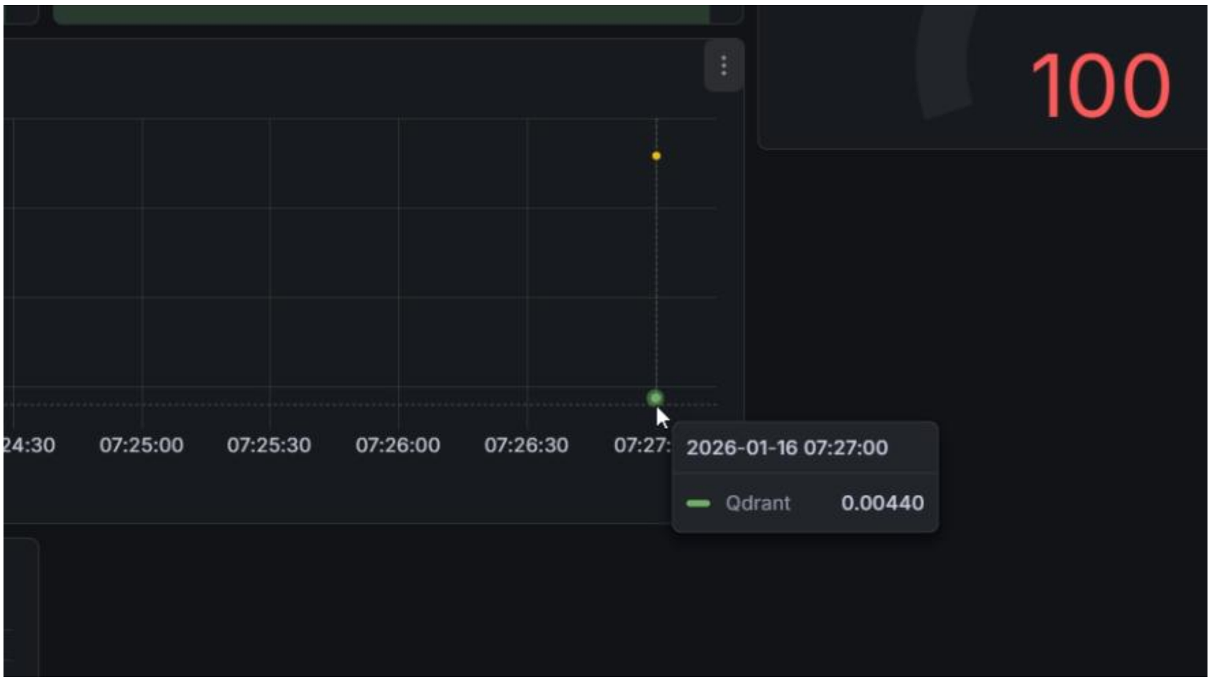

Home › Dashboards › FARO RAG Pipeline Performance
Total Vectors (Postgres)
15
Total Vectors (Qdrant)
15
Result Overlap (Accuracy Proxy)
100
Vector Search Latency (Comparison)
0.02
0.015
0.01
0.005
07:22:30
07:23:00
07:23:30
07:24:00
07:24:30
07:25:00
07:25:30
07:26:00
07:26:30
07:27:00
Qdrant
Postgres
Postgres Active Connections
1
0.75
0.5
0.25
0
07:23:00
07:24:00
07:25:00
07:26:00
07:27:00
pg_stat_activity_count{application_name="PostgreSQL JDBC Dr
pg_stat_activity_count{backend_type="client backend", contain

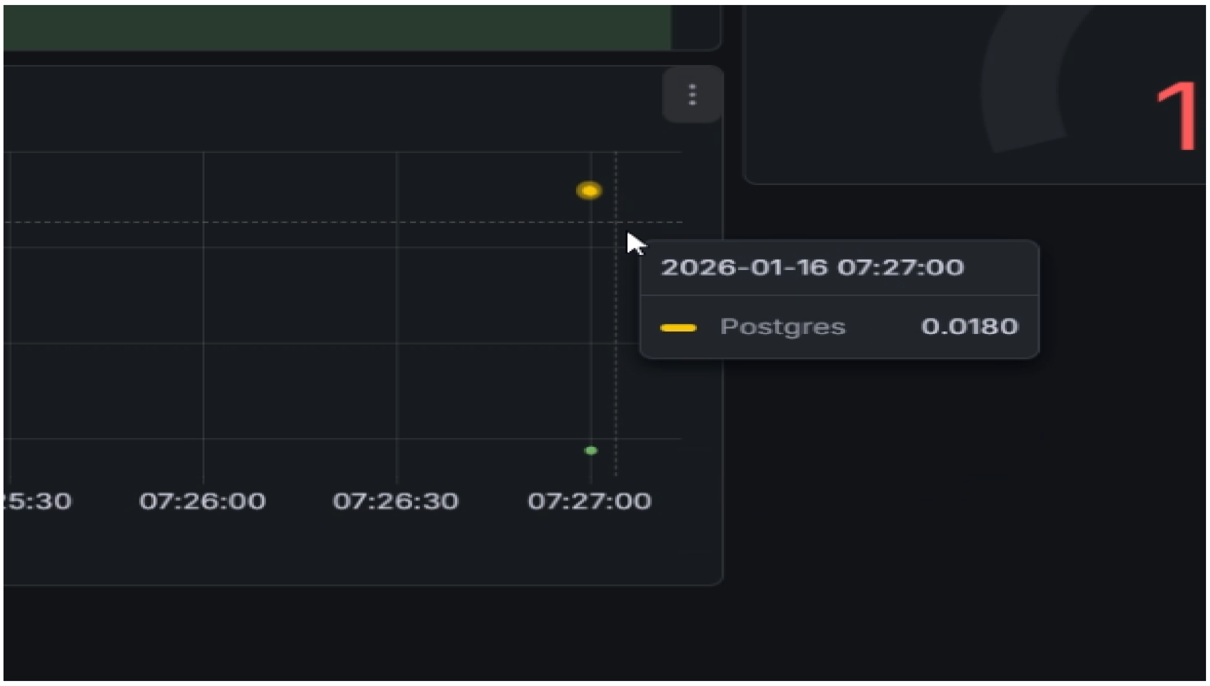
Total Vectors Panels: These stat panels show the current count of vectors in PostgreSQL and Qdrant. The PostgreSQL count comes from the pg_stat_user_tables_n_live_tup metric which reports the number of live rows in the embeddings table. The Qdrant count comes from the collection_vectors metric exposed by Qdrant's metrics endpoint. These should always be equal if both systems are in sync.

Vector Search Latency Panel: This time series graph compares search performance between Qdrant and PostgreSQL. It calculates the average duration per search by dividing the sum of all search durations by the count of searches over a 1-minute window. This uses Prometheus's rate function which handles counter resets properly. We expect Qdrant to be significantly faster than PostgreSQL because it uses specialized vector indexes.

Result Overlap Panel: This gauge shows the percentage of search results that appear in both Qdrant and PostgreSQL when running the same query. High overlap (80%+) indicates both databases are returning similar results, which means they're consistent. Low overlap suggests data synchronization issues or one database being out of date. The color thresholds provide quick visual feedback: green is good, yellow is concerning, red requires investigation.

Postgres Active Connections Panel: This tracks how many active connections are open to the PostgreSQL database. If this number grows continuously, it might indicate connection leaks in the application. If it drops to zero, it means the service lost database connectivity.





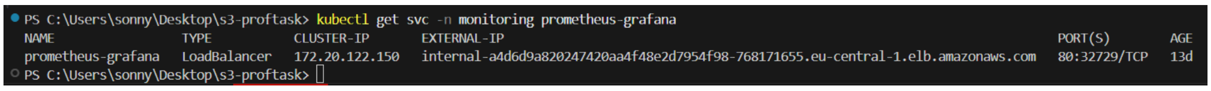
As you can see in the screenshots above, the Qdrant database has a significantly lower latency compared to the Postgres database. This is because, as mentioned before, the Qdrant database is specifically optimized for vectors, while the Postgres database is a general purpose database using a pgvector extension.

5.8 Grafana Access Configuration

Grafana is accessible through an internal AWS load balancer that gets a private IP address within the VPC.

Access method:

Get Grafana URL (after connecting to VPN)



```
PS C:\Users\sonny\Desktop\s3-proftask> kubectl get svc -n monitoring prometheus-grafana
NAME                 TYPE           CLUSTER-IP       EXTERNAL-IP                                                                 PORT(S)        AGE
prometheus-grafana   LoadBalancer   172.20.122.150   internal-a4d6d9a820247420aa4f48e2d7954f98-768171655.eu-central-1.elb.amazonaws.com   80:32729/TCP   13d
PS C:\Users\sonny\Desktop\s3-proftask>
```

Login credentials:

- Username: admin
- Password: admin123

The internal load balancer ensures that Grafana is only accessible from within the VPC. You must be connected to the VPN to reach it. The Kubernetes service name prometheus-grafana is automatically created by the Helm chart. In a production environment, you would use a more secure password and enable authentication through AWS SSO or LDAP.

5.9 Metrics Collection Architecture

The monitoring system uses a pull-based architecture where Prometheus actively scrapes metrics from targets. Here's how the data flows:

1. **Service Instrumentation:** The RAG Query service includes Prometheus client libraries that expose metrics on the /metrics endpoint
2. **ServiceMonitor Discovery:** ServiceMonitor resources tell Prometheus which services to scrape
3. **Metric Scraping:** Prometheus scrapes all targets every 15 seconds and stores time-series data
4. **Data Retention:** Prometheus stores metrics for 30 days with automatic compaction
5. **Visualization:** Grafana queries Prometheus and displays metrics on dashboards
6. **Alerting:** (Not implemented yet) Alertmanager can send notifications when metrics exceed thresholds

This architecture scales well because Prometheus handles thousands of time series efficiently. The pull model is also more reliable than push-based systems because Prometheus can detect when targets become unavailable and alert on that.

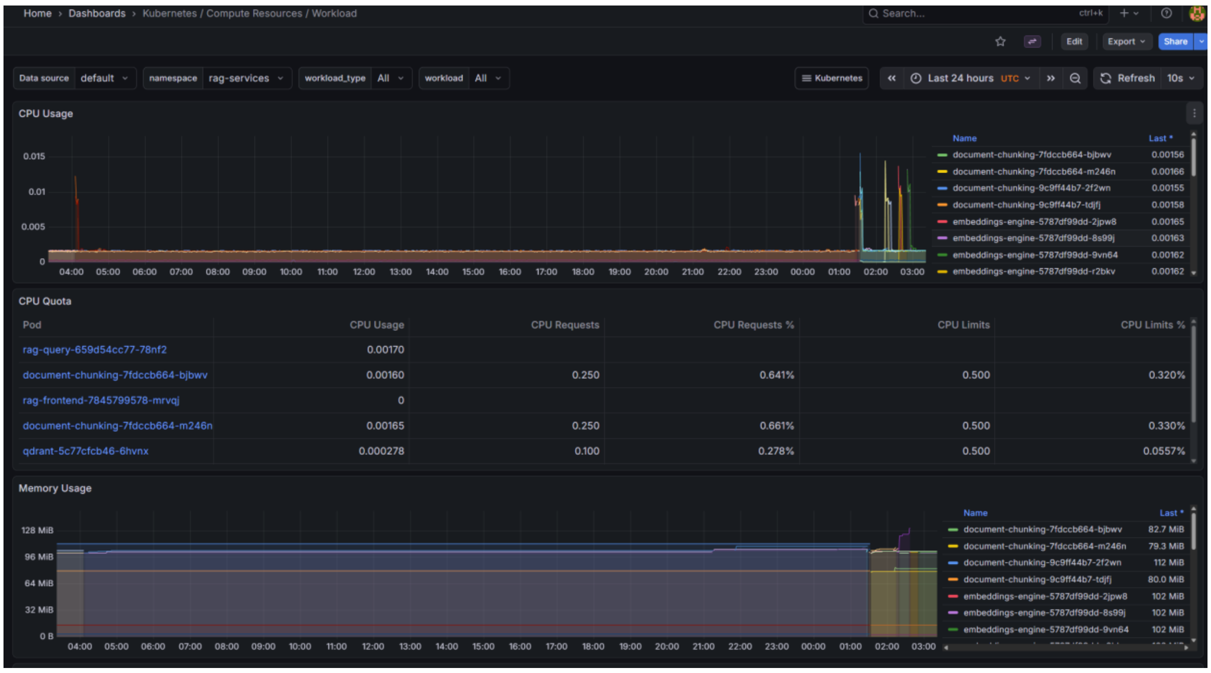
5.10 Kubernetes Workload Monitoring Dashboard

The kube-prometheus-stack Helm chart includes pre-built dashboards for monitoring Kubernetes resources. I use the "Kubernetes / Compute Resources / Workload" dashboard to track how much CPU and memory the RAG services are using.

5.10.1 Accessing the Dashboard

After deploying the Helm chart, the dashboard is automatically available in Grafana:

1. Open Grafana (via the internal load balancer URL)
2. Go to **Dashboards → Browse**
3. Select **Kubernetes / Compute Resources / Workload**
4. Choose namespace: rag-services
5. Choose workload: embeddings-engine, document-chunking, or rag-query



Data source: default | namespace: rag-services | workload_type: All | workload: All

CPU Usage

	Name	Last *
🟩	document-chunking-7fdccb664-bjbwv	0.00156
🟨	document-chunking-7fdccb664-m246n	0.00166
🟦	document-chunking-9c9ff44b7-2f2wn	0.00155
🟧	document-chunking-9c9ff44b7-tdjfj	0.00158
🟥	embeddings-engine-5787df99dd-2jpw8	0.00165
🟪	embeddings-engine-5787df99dd-8s99j	0.00163
🟩	embeddings-engine-5787df99dd-9vn64	0.00162
🟨	embeddings-engine-5787df99dd-r2bkv	0.00162

CPU Quota

Pod	CPU Usage	CPU Requests	CPU Requests %	CPU Limits	CPU Limits %
rag-query-659d54cc77-78nf2	0.00170				
document-chunking-7fdccb664-bjbwv	0.00160	0.250	0.641%	0.500	0.320%
rag-frontend-7845799578-mrvqj	0				
document-chunking-7fdccb664-m246n	0.00165	0.250	0.661%	0.500	0.330%
qdrant-5c77cfcb46-6hvnx	0.000278	0.100	0.278%	0.500	0.0557%

Memory Usage

	Name	Last *
🟩	document-chunking-7fdccb664-bjbwv	82.7 MiB
🟨	document-chunking-7fdccb664-m246n	79.3 MiB
🟦	document-chunking-9c9ff44b7-2f2wn	112 MiB
🟧	document-chunking-9c9ff44b7-tdjfj	80.0 MiB
🟥	embeddings-engine-5787df99dd-2jpw8	102 MiB
🟪	embeddings-engine-5787df99dd-8s99j	102 MiB
🟩	embeddings-engine-5787df99dd-9vn64	102 MiB

5.10.2 What the Dashboard Shows

The dashboard displays several important metrics for each service:

CPU and Memory Usage: Shows how much CPU and memory each pod is actually using compared to the limits we set. This helps identify if services need more resources.

Network Traffic: Displays data transfer between pods and databases. Useful for spotting connectivity issues or unusually high traffic.

Pod Status: Shows if pods are running, restarting, or crashing. Frequent restarts usually mean something is wrong.

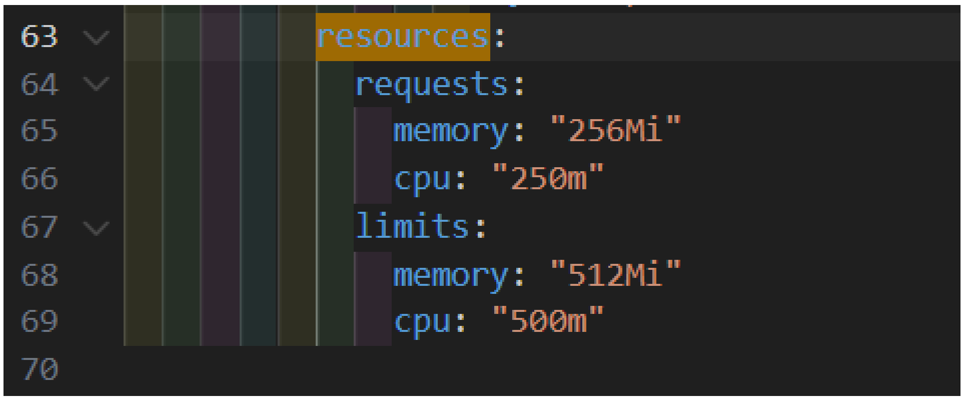
5.10.3 Why this matters for the embeddings engine

The embeddings-engine uses the most resources because it processes text and generates vectors. The dashboard helps spot problems:

If CPU usage constantly hits the 500m limit, the service gets slowed down (throttled) and takes longer to process documents. If memory approaches 512Mi, Kubernetes might kill the pod to prevent it from using too much memory. If you see frequent restarts, it usually means the pod crashed due to memory issues or lost database connectivity.

5.10.4 Current resource limits

Here's what I configured for the embeddings-engine:



```yaml
resources:
  requests:
    memory: "256Mi"
    cpu: "250m"
  limits:
    memory: "512Mi"
    cpu: "500m"
```

The "requests" tell Kubernetes the minimum resources needed to run the pod. The "limits" prevent the pod from using too much and affecting other services. I chose these values because the service needs memory for database connections and buffering text chunks, plus CPU for making API calls to Portkey.

6. Security Configuration

6.1 Overview

Security groups act as virtual firewalls that control network traffic to and from AWS resources. I configured six main security groups for the RAG infrastructure: one for the VPN Bastion, one for Kubernetes worker nodes, one for the Public Load Balancer, one for the RDS database, one for the Qdrant Vector DB, and one for Monitoring services. Each security group follows the principle of least privilege by only allowing necessary traffic.

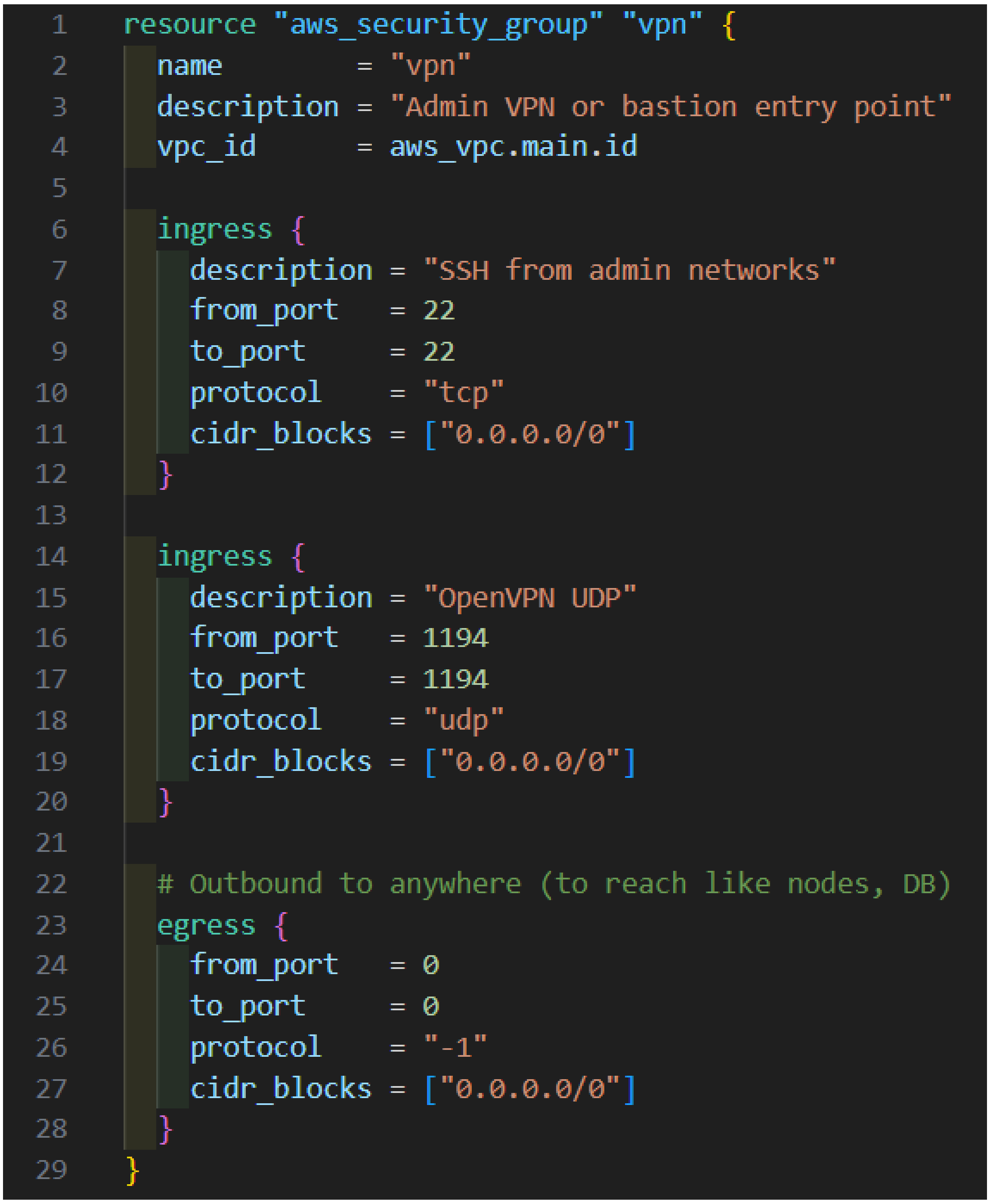
6.2 VPN / Bastion Security Group

The VPN security group protects the entry point for administrators. It acts as a "Jump Host," allowing secure tunneling into the private network to access internal dashboards and databases.

Security Group Rules:

Direction	**Source/Destination**	**Port**	**Protocol**	**Reason**
Inbound	0.0.0.0/0 (Internet)	22	TCP	Allow SSH access for administrators to the Bastion host
Inbound	0.0.0.0/0 (Internet)	1194	UDP	Allow OpenVPN client connections to tunnel into the network
Outbound	0.0.0.0/0 (All)	All	All	Allow the VPN server to route traffic to internal private subnets

Code snippet from security group configuration:



```
resource "aws_security_group" "vpn" {
  name        = "vpn"
  description = "Admin VPN or bastion entry point"
  vpc_id      = aws_vpc.main.id

  ingress {
    description = "SSH from admin networks"
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  ingress {
    description = "OpenVPN UDP"
    from_port   = 1194
    to_port     = 1194
    protocol    = "udp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  # Outbound to anywhere (to reach like nodes, DB)
  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

Port 22 is open for SSH management, while port 1194 (UDP) is the standard port for OpenVPN traffic. By allowing these connections, administrators can establish a secure tunnel. Once inside, the VPN security group ID (aws_security_group.vpn.id) is used as a "key" to unlock access to other internal resources.

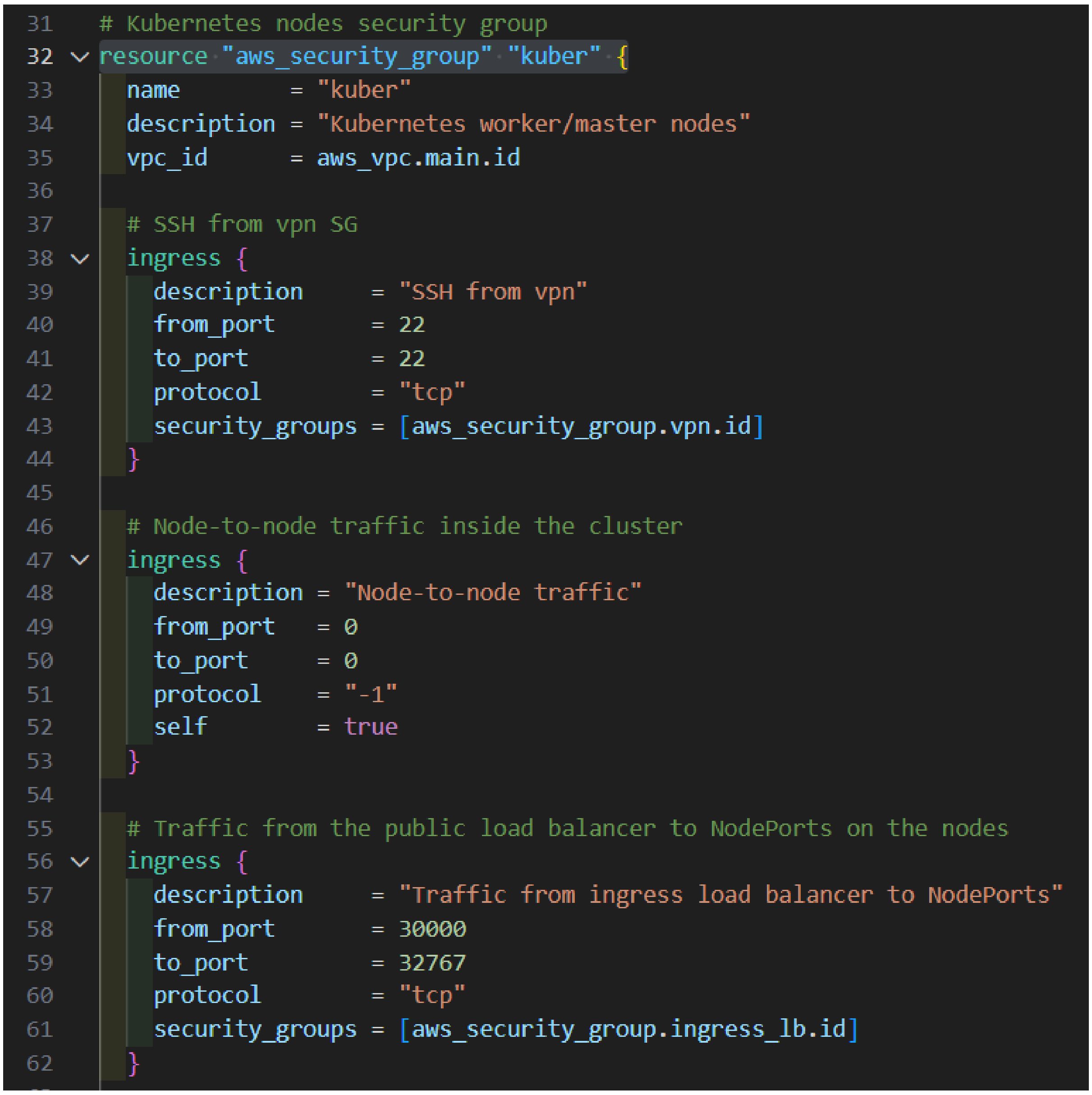
6.3 Kubernetes Nodes Security Group

The worker nodes security group (kuber) controls traffic to and from the EC2 instances that run the application pods.

Security Group Rules:

Direction	**Source/Destination**	**Port**	**Protocol**	**Reason**
Inbound	VPN Security Group	22	TCP	Allow secure SSH access from the Bastion host (no direct internet)
Inbound	Self (same SG)	All	All	Allow pods to communicate with each other across different nodes
Inbound	Load Balancer SG	30000-32767	TCP	Allow Public Load Balancer to forward traffic to NodePorts
Outbound	0.0.0.0/0 (All)	All	All	Allow nodes to pull images from ECR and reach external APIs

Code snippet from security group configuration:



```
# Kubernetes nodes security group
resource "aws_security_group" "kuber" {
  name        = "kuber"
  description = "Kubernetes worker/master nodes"
  vpc_id      = aws_vpc.main.id

  # SSH from vpn SG
  ingress {
    description     = "SSH from vpn"
    from_port       = 22
    to_port         = 22
    protocol        = "tcp"
    security_groups = [aws_security_group.vpn.id]
  }

  # Node-to-node traffic inside the cluster
  ingress {
    description = "Node-to-node traffic"
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    self        = true
  }

  # Traffic from the public load balancer to NodePorts on the nodes
  ingress {
    description     = "Traffic from ingress load balancer to NodePorts"
    from_port       = 30000
    to_port         = 32767
    protocol        = "tcp"
    security_groups = [aws_security_group.ingress_lb.id]
  }
```

The self-referencing rule ensures that microservices running on different nodes can talk to each other. SSH is strictly limited to the VPN security group, preventing public attacks. The range 30000-32767 is the standard Kubernetes NodePort range, which the public Load Balancer uses to route user requests into the cluster.

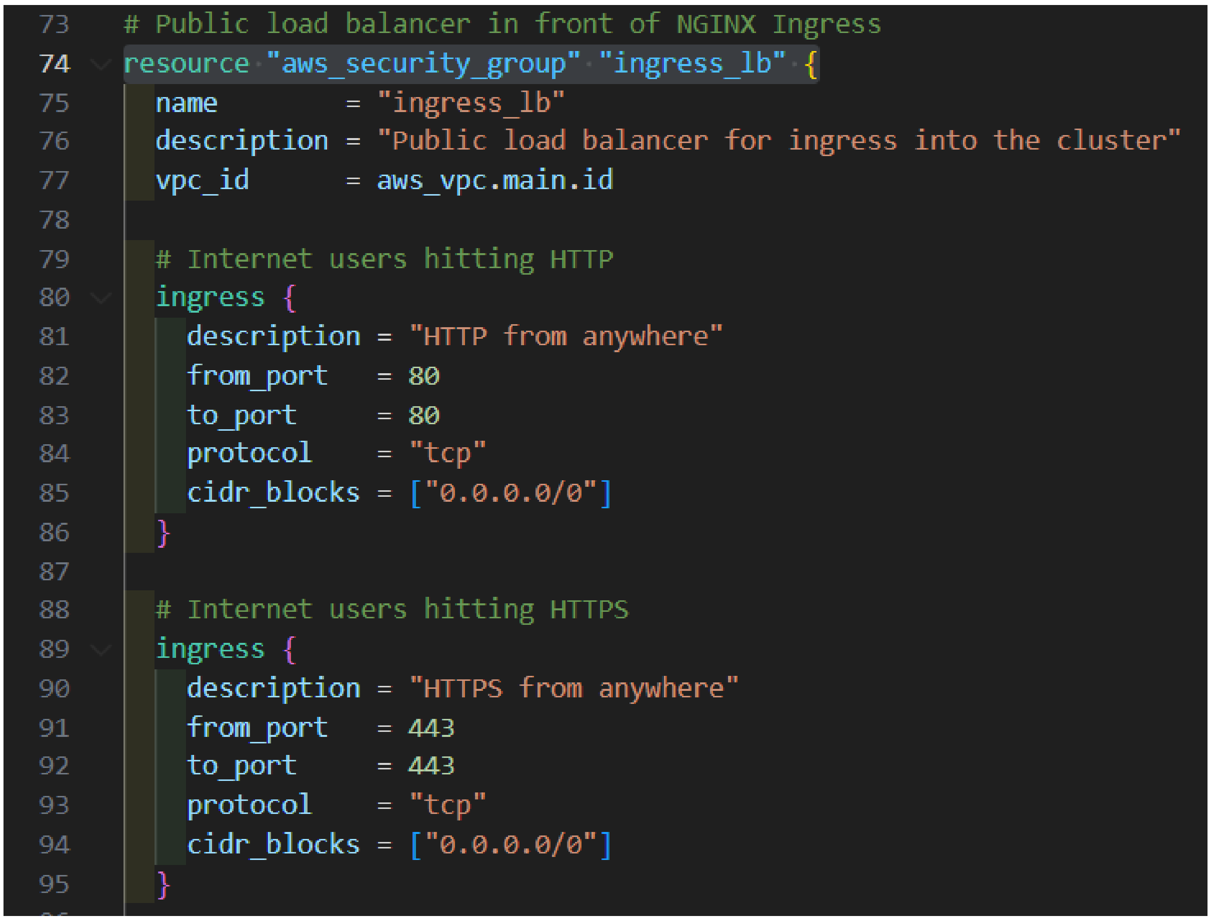
6.4 Load Balancer Security Group

The Ingress Load Balancer security group manages public web access to the application.

Security Group Rules:

Direction	Source/Destination	Port	Protocol	Reason
Inbound	0.0.0.0/0 (Internet)	80	TCP	Allow public HTTP traffic
Inbound	0.0.0.0/0 (Internet)	443	TCP	Allow public HTTPS traffic
Outbound	0.0.0.0/0 (All)	All	All	Allow forwarding requests to the backend worker nodes

Code snippet from security group configuration:



```
# Public load balancer in front of NGINX Ingress
resource "aws_security_group" "ingress_lb" {
  name        = "ingress_lb"
  description = "Public load balancer for ingress into the cluster"
  vpc_id      = aws_vpc.main.id

  # Internet users hitting HTTP
  ingress {
    description = "HTTP from anywhere"
    from_port   = 80
    to_port     = 80
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  # Internet users hitting HTTPS
  ingress {
    description = "HTTPS from anywhere"
    from_port   = 443
    to_port     = 443
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
```

This is the only security group with open web ports (80/443). It acts as the front door, accepting user traffic and passing it securely to the private worker nodes.

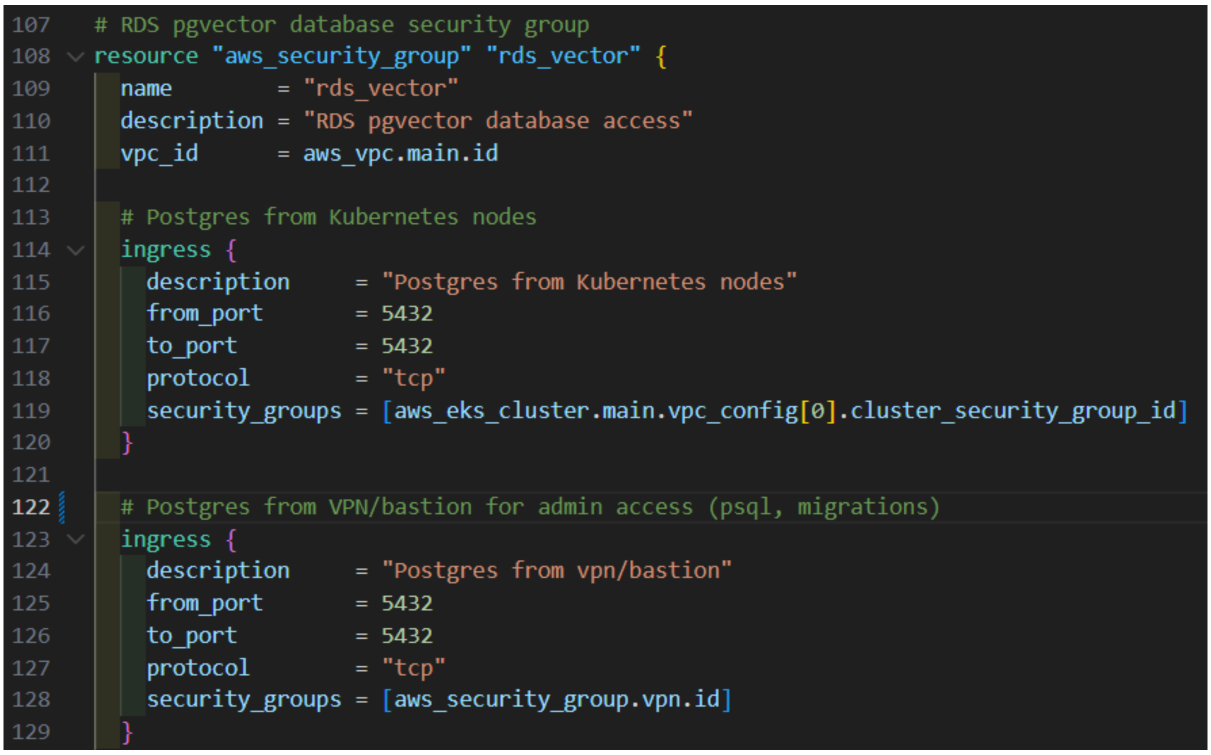
6.5 RDS PostgreSQL Security Group

The RDS security group controls access to the PostgreSQL database used for "Shadow Storage" of vectors.

Security Group Rules:

Direction	Source/Destination	Port	Protocol	Reason
Inbound	Cluster Security Group	5432	TCP	Allow RAG microservices to read/write embeddings
Inbound	VPN Security Group	5432	TCP	Allow admins to run SQL queries via VPN for debugging
Outbound	0.0.0.0/0 (All)	All	All	Allow database to respond to queries

Code snippet from security group configuration:



```
# RDS pgvector database security group
resource "aws_security_group" "rds_vector" {
  name        = "rds_vector"
  description = "RDS pgvector database access"
  vpc_id      = aws_vpc.main.id

  # Postgres from Kubernetes nodes
  ingress {
    description     = "Postgres from Kubernetes nodes"
    from_port       = 5432
    to_port         = 5432
    protocol        = "tcp"
    security_groups = [aws_eks_cluster.main.vpc_config[0].cluster_security_group_id]
  }

  # Postgres from VPN/bastion for admin access (psql, migrations)
  ingress {
    description     = "Postgres from vpn/bastion"
    from_port       = 5432
    to_port         = 5432
    protocol        = "tcp"
    security_groups = [aws_security_group.vpn.id]
  }
```

Direct access is blocked from the internet. Only the application pods (via the Cluster SG) and administrators (via the VPN SG) can connect to the database on port 5432.

6.6 Qdrant Vector DB Security Group

The Qdrant security group protects the EC2 instance hosting the primary vector database.

Security Group Rules:

Direction	**Source/Destination**	**Port**	**Protocol**	**Reason**
Inbound	Cluster Security Group	6333	TCP	Allow RAG apps to search vectors (REST API)
Inbound	Cluster Security Group	6334	TCP	Allow high-performance data transfer (gRPC API)
Inbound	VPN Security Group	6333	TCP	Allow admins to view the Qdrant Web UI Dashboard
Inbound	VPN Security Group	22	TCP	Allow SSH access for maintenance
Outbound	0.0.0.0/0 (All)	All	All	Allow instance to pull updates

Code snippet from security group configuration:

```
# Security group for Qdrant
resource "aws_security_group" "qdrant" {
  name        = "qdrant"
  description = "Qdrant vector database access"
  vpc_id      = aws_vpc.main.id

  # Qdrant REST API from Kubernetes nodes
  ingress {
    description     = "Qdrant REST API from K8s"
    from_port       = 6333
    to_port         = 6333
    protocol        = "tcp"
    security_groups = [aws_eks_cluster.main.vpc_config[0].cluster_security_group_id]
  }

  # Qdrant gRPC API from Kubernetes nodes (optional, for better performance)
  ingress {
    description     = "Qdrant gRPC from K8s"
    from_port       = 6334
    to_port         = 6334
    protocol        = "tcp"
    security_groups = [aws_eks_cluster.main.vpc_config[0].cluster_security_group_id]
  }

  # SSH from VPN for admin access
  ingress {
    description     = "SSH from vpn"
    from_port       = 22
    to_port         = 22
    protocol        = "tcp"
    security_groups = [aws_security_group.vpn.id]
  }

  # Qdrant API from VPN for testing
  ingress {
    description     = "Qdrant API from vpn"
    from_port       = 6333
    to_port         = 6333
    protocol        = "tcp"
    security_groups = [aws_security_group.vpn.id]
  }

  ingress {
    description     = "Qdrant REST API from Worker Nodes"
    from_port       = 6333
    to_port         = 6333
    protocol        = "tcp"
    security_groups = [aws_security_group.kuber.id]
  }
```

Access is strictly segmented. The application uses ports 6333/6334 for operations, while administrators use the VPN to access the Web UI on port 6333 for visual verification of vector data.

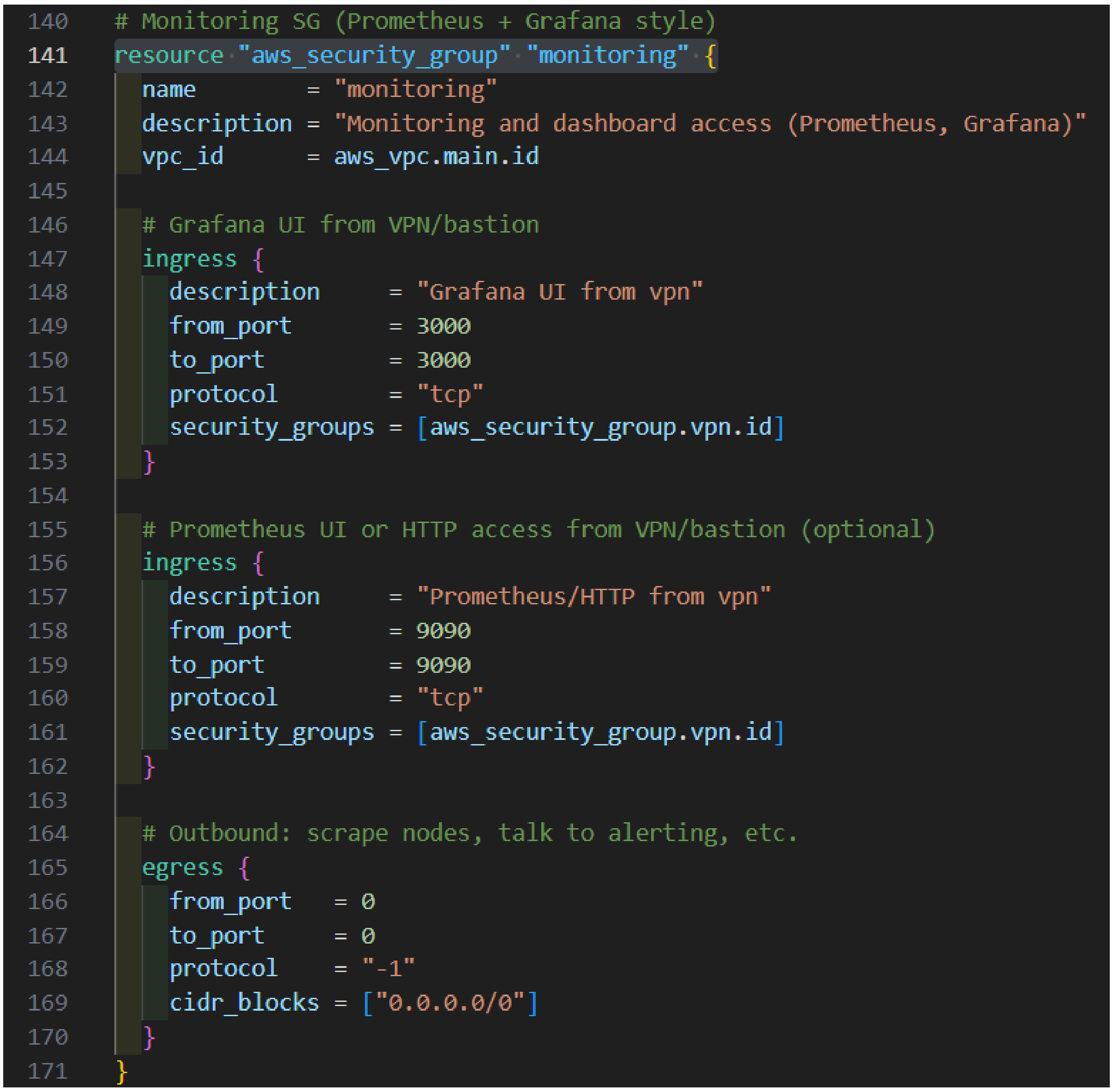
6.7 Monitoring Security Group

The monitoring security group secures the internal observability stack (Grafana and Prometheus).

Security Group Rules:

Direction	Source/Destination	Port	Protocol	Reason
Inbound	VPN Security Group	3000	TCP	Allow VPN users to access Grafana Dashboards
Inbound	VPN Security Group	9090	TCP	Allow VPN users to access Prometheus UI
Outbound	0.0.0.0/0 (All)	All	All	Allow Prometheus to scrape metrics from all targets

Code snippet from security group configuration:



```
# Monitoring SG (Prometheus + Grafana style)
resource "aws_security_group" "monitoring" {
  name        = "monitoring"
  description = "Monitoring and dashboard access (Prometheus, Grafana)"
  vpc_id      = aws_vpc.main.id

  # Grafana UI from VPN/bastion
  ingress {
    description     = "Grafana UI from vpn"
    from_port       = 3000
    to_port         = 3000
    protocol        = "tcp"
    security_groups = [aws_security_group.vpn.id]
  }

  # Prometheus UI or HTTP access from VPN/bastion (optional)
  ingress {
    description     = "Prometheus/HTTP from vpn"
    from_port       = 9090
    to_port         = 9090
    protocol        = "tcp"
    security_groups = [aws_security_group.vpn.id]
  }

  # Outbound: scrape nodes, talk to alerting, etc.
  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

Monitoring tools often contain sensitive system information. By restricting ports 3000 and 9090 to the VPN Security Group, I ensured that these dashboards are completely invisible to the public internet. It is only accessible if you're connected to the VPN.

6.8 Security Best Practices Applied

- **Principle of Least Privilege:** Each security group only allows the minimum necessary traffic.
- **Defense in Depth:** Multiple layers of security protect the infrastructure. Network-level security groups block unauthorized traffic, IAM roles control what resources can be accessed, and SSL encryption protects data in transit.
- **Bastion-First Architecture:** The VPN/Bastion serves as the single point of entry for management. No other resource accepts SSH from the internet.

- **Private subnet isolation:** Critical resources (RDS, Qdrant, K8s Nodes) reside in private subnets and are protected by Security Groups that reject all public inbound traffic.
- **Role-Based Access:** Security groups reference each other by ID (e.g., allowing traffic from aws_security_group.vpn.id) rather than IP ranges, creating a dynamic and secure trust relationship between resources.

7. Research & technical decision justification(DOT Framework)

7.1 Overview

This section documents the research I conducted to make informed technical decisions for the RAG pipeline infrastructure. I used the DOT framework to structure my research, combining Literature Study to understand available options and Best Good and Bad Practices to learn from real-world implementations. Each major decision is justified based on the requirements and supported by research findings.

7.2 Research Methodology (DOT Framework)

I applied two research methods from the DOT framework to investigate monitoring solutions and architectural choices:

Literature Study: I reviewed official documentation, technical blogs, and comparison articles to understand the capabilities and limitations of different monitoring solutions. This helped me identify the features and trade-offs of each option.

Best Good and Bad Practices: I analyzed case studies and experience reports from companies running similar infrastructure. This revealed common pitfalls and proven patterns for production deployments.

7.3 Monitoring Solution Research

7.3.1 Research Question

Which monitoring solution best fits the requirements for observing a RAG pipeline running on EKS with external components like Qdrant and RDS?

7.3.2 Requirements

- Monitor Kubernetes workloads (pods, deployments, resource usage)
- Collect custom application metrics (search latency, vector counts)
- Scrape metrics from external services (Qdrant on EC2, RDS PostgreSQL)
- Visualize data with customizable dashboards
- Cost-effective for a student project

7.3.3 Monitoring Options Comparison

Solution	Deployment	Kubernetes Support	Custom Metrics	External Targets	Cost	Ease of Setup	Sources
Prometheus + Grafana	Self-hosted on EKS	Excellent (native)	Yes (Pull model)	Yes (any HTTP endpoint)	Free (infrastructure only)	Medium (Helm chart available)	(Prometheus, 2025) (Grafana, 2025
AWS CloudWatch	Managed AWS service	Good (Container Insights)	Yes (agent required)	Limited (AWS services only)	Pay per metric	Easy (native integration)	(AWS, Amazon CloudWatch Container Insights, 2025) (AWS, CloudWatch Pricing, 2025)
NagiosXI	Self-hosted	Basic (plugins required)	Yes (NRPE/NCPA)	Yes (agent-based)	Free (Core) / $1,995+ (XI)	Hard (manual config)	(Nagios, Nagios Documentatio 2025) (Nagios, Nagios Pricing, 2025)
ELK Stack	Self-hosted	Good (Metricbeat)	Yes	Yes (Beats agents)	Free (infrastructure only)	Hard (multiple components)	(Elastic, 2025)

7.3.4 Decision: Prometheus + Grafana

I chose Prometheus and Grafana deployed via Helm chart for the following reasons:

Native Kubernetes integration: Prometheus was designed for Kubernetes environments. It automatically discovers services through Kubernetes APIs using ServiceMonitor resources. The kube-prometheus-stack Helm chart includes pre-configured dashboards for node metrics, pod performance, and cluster health, which saved significant setup time (Prometheus, 2025).

Flexible metric collection: Prometheus uses a pull model that works with any service exposing HTTP metrics endpoints. This flexibility allowed me to monitor the external Qdrant EC2 instance by creating a Kubernetes Service bridge. CloudWatch Container Insights doesn't support pulling metrics from external EC2 instances in the same way (AWS, Amazon CloudWatch Container Insights, 2025). NagiosXI would require installing and configuring NRPE agents on each target, adding complexity (Nagios, Nagios XI Documentation, 2025).

Custom application metrics: The RAG Query service uses prometheus-client libraries to expose custom metrics like search latency per database and result overlap. Prometheus scrapes these directly without requiring additional agents or configuration. CloudWatch would require CloudWatch agent installation and custom metric publishing code (AWS, CloudWatch Pricing, 2025).

Cost: For a student project, Prometheus and Grafana are completely free. We only pay for the infrastructure they run on. CloudWatch charges $0.30 per custom metric per month, which would add up quickly with hundreds of metrics (AWS, Amazon CloudWatch Container Insights, 2025). NagiosXI commercial version costs $1,995+ for licensing (Nagios, Nagios XI Pricing, 2025).

Dashboard customization: Grafana's dashboard JSON format allows version-controlled custom dashboards deployed as ConfigMaps. I created a RAG-specific dashboard tracking vector counts and search performance. CloudWatch dashboards are less flexible and can't be version controlled as easily (AWS, CloudWatch Pricing, 2025). Nagios dashboards are primarily status-focused rather than metric visualization (Nagios, Nagios XI Documentation, 2025).

Community and learning: Prometheus is the de-facto standard for Kubernetes monitoring. Learning it provides valuable skills for the job market. According to the CNCF Survey 2023, Prometheus is used by 70% of organizations running Kubernetes (Foundation, 2023).

Kubernetes-native tooling: The ELK Stack requires deploying and managing multiple components (Elasticsearch, Logstash, Kibana, Metricbeat) which increases operational complexity. Prometheus and Grafana are purpose-built for metrics and provide a simpler architecture for this use case (Elastic, 2025).

7.3.5 Trade-offs Accepted

Operational overhead: Self-hosting Prometheus requires maintaining the infrastructure and managing storage. Managed solutions like CloudWatch handle this automatically. However, the kube-prometheus-stack Helm chart significantly reduces this burden by automating deployment and configuration.

Limited alerting: Prometheus's Alertmanager is functional but less sophisticated than commercial solutions. For this project, basic alerting was sufficient, but production environments might need more advanced features like incident management integrations.

Storage management: Prometheus stores 30 days of metrics by default. Longer retention requires additional storage configuration or integration with systems like

Thanos. CloudWatch automatically handles long-term storage (AWS, CloudWatch Pricing, 2025).

7.4 Portkey API Integration

7.4.1 Decision Rationale

Our teacher provided a Portkey API key for the class rather than having students manage individual embedding provider accounts. This decision offers several benefits supported by research:

Unified gateway: Portkey provides a single API endpoint that routes to multiple embedding providers (OpenAI, Cohere, etc.). This abstraction simplifies switching between models without code changes (Portkey, 2025). If OpenAI's API experiences downtime, Portkey can failover to alternative providers automatically.

Cost management: For educational settings, Portkey allows instructors to set spending limits and monitor usage across all students. According to Portkey's documentation, this prevents the "bill shock" scenarios where students accidentally run up large API costs (Portkey, 2025).

Observability: Portkey tracks metrics like request count, latency, and error rates for all API calls. This visibility helps debug issues when embeddings fail to generate. Direct provider APIs would require custom instrumentation to achieve the same observability.

Learning environment optimization: The managed nature of Portkey reduces setup complexity, allowing students to focus on infrastructure and architecture rather than API key management. This aligns with the course's learning objectives around cloud infrastructure.

7.4.2 Trade-off: Vendor Lock-in

The downside is dependency on Portkey's API format. If Portkey shuts down or changes pricing dramatically, migration to direct provider APIs requires code changes. However, for a time-limited educational project, this risk is acceptable given the convenience benefits.

7.5 EKS Instance Type Selection

7.5.1 Decision: t3.medium Instances

I chose t3.medium instances (2 vCPU, 4GB RAM) for the EKS node group based on workload analysis and AWS documentation.

Resource requirements: Each RAG service requires approximately 500MB-1GB memory for normal operation. With 4GB per node, we can run 2-3 services per node with overhead for Kubernetes system components (kubelet, kube-proxy) which typically consume 500-800MB (AWS, Amazon EKS Best Practices Guide for Cost Optimization, 2025).

Burstable performance: T3 instances use CPU credits to handle burst workloads. The embeddings-engine experiences CPU spikes during batch processing but spends most time idle waiting for API responses. This burst pattern matches T3's design perfectly (AWS, Amazon EC2 T3 Instances, 2025). M-series instances with constant performance would cost 30-40% more without providing benefits.

Cost optimization: At $0.0416/hour per instance ($30/month), three t3.medium nodes cost approximately $90/month. Equivalent t3.large instances (4 vCPU, 8GB) would cost $180/month without significant performance improvement for this workload (AWS, Amazon EC2 T3 Instances, 2025).

AWS recommendations: According to AWS best practices for EKS, t3.medium is recommended for development and small production workloads with fewer than 30 pods per cluster (AWS, Amazon EKS Best Practices Guide for Cost Optimization, 2025). The RAG pipeline runs 6-8 pods total, well within this guideline.

7.6 References

AWS. (2025). *Amazon CloudWatch Container Insights*. From Amazon AWS: https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/ContainerInsights.html

AWS. (2025). *Amazon EC2 T3 Instances*. From AWS Amazon: https://aws.amazon.com/ec2/instance-types/t3/

AWS. (2025). *Amazon EKS Best Practices Guide for Cost Optimization*. From AWS Amazon: https://aws.github.io/aws-eks-best-practices/cost_optimization/

AWS. (2025). *CloudWatch Pricing*. From AWS Amazon: https://aws.amazon.com/cloudwatch/pricing/

Elastic. (2025). *Metricbeat: Lightweight Shipper for Metrics*. From Elastic: https://www.elastic.co/beats/metricbeat

Foundation, C. N. (2023). *CNCF Annual Survey 2023*. From CNCF: https://www.cncf.io/reports/cncf-annual-survey-2023/

Grafana. (2025). *Grafana documentation*. From Grafana: https://grafana.com/docs/grafana/latest/

Nagios. (2025). *Nagios XI Documentation*. From Nagios: https://www.nagios.org/documentation/

Nagios. (2025). *Nagios XI Pricing*. From Nagios: https://www.nagios.com/products/nagios-xi/

Portkey. (2025). *Portkey Documentation*. From Portkey: https://portkey.ai/docs

Prometheus. (2025). *Overview - Prometheus*. From Prometheus: https://prometheus.io/docs/introduction/overview/

8. Planning & Project Management

8.1 Initial Task Breakdown

When I received my assigned tasks for the proftaak, I broke them down into smaller, manageable pieces. I used a simple task list approach rather than formal project management tools because the scope was limited to my individual contributions.

8.1.1 My main tasks:

1. Set up EKS cluster infrastructure
2. Implement embeddings engine service
3. Deploy monitoring stack
4. Configure security groups
5. Write documentation

I further broke these down into subtasks:

EKS Cluster (Week 1-2):

- Research EKS setup requirements and best practices
- Create Terraform configurations for cluster and node group
- Set up IAM roles for cluster and nodes
- Configure OIDC provider for IRSA
- Test cluster access with kubectl
- Create service account with S3 permissions

Embeddings Engine (Week 2-3):

- Design dual-storage integration points
- Implement health check endpoint
- Build /embed endpoint for single embeddings
- Build /process/s3 endpoint for batch processing
- Add PostgreSQL initialization logic
- Create Kubernetes deployment manifest
- Test with sample documents

Monitoring Stack (Week 3-4):

- Research monitoring options (Prometheus vs alternatives)
- Deploy Prometheus and Grafana via Helm
- Configure ServiceMonitor for Qdrant
- Configure ServiceMonitor for RAG Query service
- Set up PostgreSQL exporter
- Create custom RAG dashboard
- Test metric collection

Security Configuration (Week 4):

- Document current security group rules
- Review and tighten unnecessary permissions
- Create security group reference tables
- Verify connectivity after changes

Documentation (Week 5):

- Write technical sections
- Add code snippets and explanations
- Create screenshot placeholders
- Write reflection sections
- **Timeline and Milestones**

I created a rough timeline with milestones to track major deliverables:

Week 1 (Oct 28 - Nov 3):

- Milestone: EKS cluster operational and accessible via kubectl
- Actual: Completed on schedule

Week 2 (Nov 4 - Nov 10):

- Milestone: Embeddings engine deployed and processing test documents
- Actual: Delayed by 2 days due to Qdrant connectivity issues

Week 3 (Nov 11 - Nov 17):

- Milestone: Monitoring stack collecting metrics from all services

- Actual: Completed slightly ahead of schedule

Week 4 (Nov 18 - Nov 24):

- Milestone: Security configuration documented and verified
- Actual: Completed on schedule

Week 5 (Nov 25 - Dec 1):

- Milestone: Documentation complete
- Actual: In progress

8.2 Progress tracking

I tracked progress using a simple checklist in a text file in my project repository. Every day I would review what I completed and what remained. This worked well for individual work because it was lightweight and didn't require maintaining a separate project management tool.

I updated this daily and used it during standup meetings with teammates to report progress and blockers.

8.3 Changes from Initial Plan

8.3.1 Change 1: Qdrant Integration Complexity

Original plan: I assumed Qdrant would be straightforward to integrate since it exposes HTTP endpoints.

What changed: Monitoring Qdrant from Prometheus required the Service/Endpoints bridge pattern, which I hadn't planned for. This added half a day of research and implementation.

Why it changed: I underestimated the difference between monitoring in-cluster services and external EC2 instances. Kubernetes service discovery doesn't automatically work with external resources.

How I adapted: I allocated time from my "buffer week" to research and implement the bridge pattern. I also documented it clearly for future reference.

8.3.2 Change 2: Security Group Refinement

Original plan: I planned to configure security groups once during EKS setup and be done.

What changed: I had to revisit security groups three times: initial setup, after monitoring deployment, and final cleanup for documentation.

Why it changed: Each new component (monitoring, exporters) required different network access that I hadn't anticipated during initial planning.

How I adapted: I learned to test security groups incrementally. After adding a new component, I immediately verified its connectivity and adjusted rules. This prevented accumulating security issues.

8.3.3 Change 3: Documentation Scope

Original plan: I initially planned to document only the technical implementation (Sections 1-5).

What changed: My teacher's feedback indicated I needed reflection and process sections (Sections 6-10) to achieve "advanced" assessment level.

Why it changed: The assessment criteria required demonstrating research methodology, reflection, and communication skills beyond just technical implementation.

How I adapted: I allocated an extra week for documentation and prioritized writing about decision-making process and learning outcomes. This actually helped me better understand my own work.

8.4 Dealing with Blockers

8.4.1 Blocker 1: Waiting for Database Setup

At the beginning of Week 2, I needed PostgreSQL RDS to be available before I could test database connectivity in the embeddings engine. The teammate responsible for RDS setup was delayed.

Solution: I developed the embeddings logic using local PostgreSQL in Docker to continue making progress. Once RDS was available, I only had to change the connection string. This parallel development prevented idle time.

8.4.2 Blocker 2 Missing Portkey API Documentation

The Portkey API documentation was less detailed than I expected, especially regarding the embedding dimension sizes.

Solution: I tested with small requests to understand the API behavior. I also asked teammates if they had figured out the correct parameters. This collaborative troubleshooting resolved the issue faster than waiting for documentation updates.

8.5 Time Management Strategies

Priorities: Each week, I identified 2-3 "must-do" tasks that would block other work if not completed. These always got done first. Lower-priority tasks could shift to the next week if needed.

Parallel development: When possible, I worked on tasks that didn't depend on each other. For example, while waiting for database access, I worked on the monitoring stack. This maximized productivity.

Planning: I planned for 70% of available time rather than 100%. This buffer absorbed unexpected issues like the Qdrant integration complexity without derailing the entire schedule.

Regular check-ins: Weekly team meetings helped identify integration issues early. If my work would affect a teammate's component, we discussed it before implementation rather than discovering conflicts later.

8.6 Lessons Learned About Planning

Underestimating integration work: I consistently underestimated how long it takes to integrate different components. Setting up individual services is quick, but making them work together takes debugging time.

Value of incremental testing: Testing each component immediately after implementation caught issues early. Waiting until everything was complete would have made debugging much harder.

Documentation should be continuous: Writing documentation after finishing all implementation meant I forgot some details. Next time, I'll document decisions and configurations as I make them.

Importance of communication: Regular updates to teammates prevented duplicate work and revealed dependencies I hadn't considered. For example, I learned the chunking service would trigger the embeddings engine via HTTP, which influenced my endpoint design.

9. Stakeholder communication

9.1 Team Structure and Roles

The proftaak team consisted of 6 students, each responsible for different components of the RAG pipeline. I was specifically tasked with the security configuration, embeddings engine service, EKS infrastructure and the monitoring. This division meant we had to coordinate carefully because each component depended on others working correctly.

9.2 Communication Tools and Frequency

9.2.1 Whatsapp

We used Whatsapp as our primary communication platform. Whatsapp worked well because we could respond asynchronously. Not everyone was online at the same time due to different schedules, so Whatsapp's message history kept everyone informed.

Communication frequency: We posted updates daily, even if just "working on X today" or "stuck on Y, will figure it out tomorrow." This visibility helped prevent duplicate work and allowed others to offer help.

9.3 Weekly Standup Meetings

Every Monday at 10:00, we had a 30-minute video call to discuss progress and plan the week.

Meeting Structure:

1. Each person shared what they completed last week (5 minutes each)
2. Each person shared what they planned for this week
3. We discussed blockers and dependencies (10 minutes)
4. We coordinated integration testing if needed

9.4 Coordination with Team Members

9.4.1 With Mehdi (responsible for the Chunking Service)

Key Coordination Points:

S3 Event Trigger Design: We needed to agree on how the chunking service would notify the embeddings engine about new files. We discussed two options: SNS notifications or HTTP POST. We chose HTTP POST because it was simpler to implement and didn't require additional AWS resources.

API Contract: We documented the expected request format:

json

```
{
  "s3_bucket": "faro-rag-documents-eu-central-1",
  "s3_key": "chunks/uuid-here.json"
}
```

This written contract meant Mehdi could implement the trigger without waiting for my endpoint to be ready. We tested integration once both sides were complete.

Dependency Management: Mehdi's chunking service had to be deployed before I could fully test the embeddings engine. We coordinated deployment timing so I could run end-to-end tests.

9.4.2 With Awo (the Database Administrator)

Key Coordination Points:

Database Credentials: Awo created the RDS instance and Qdrant EC2 instance. We agreed to store credentials in Kubernetes Secrets rather than hardcoding them. Awo shared the secret values via Whatsapp, and I created the Kubernetes Secret.

PostgreSQL Extension: I needed the vector extension installed before my initialization code would work. Awo initially didn't know about this requirement. After I explained why it was needed, they installed it via the RDS console.

Qdrant Collection Management: We discussed whether Awo should pre-create the faro_docs collection or if my service should create it. We decided my service would handle collection creation because it knew the vector dimensions and distance metric needed.

9.4.3 With Imane (responsible for the Query Service)

Key Coordination Points:

Metrics Endpoint: Imane's RAG query service needed to expose Prometheus metrics for my monitoring dashboard. I shared the Prometheus client library they should use and explained what metrics would be useful (search latency, request count).

Health Check Format: We standardized on the same health check response format across all services:

json

```
{
  "status": "healthy",
  "database": "connected"
}
```

This consistency made monitoring easier and looked more professional.

9.5 Teacher Interactions

9.5.1 Regular Check-ins

We met with our teacher (Erik Aerdts) every two weeks to review progress and get feedback. This helped keeping track of how the group performed. Communication wise and also tasks.

9.6 Handling Disagreement

9.6.1 Disagreement: Monitoring Scope

Situation: We initially didn't want to add Prometheus metrics to the query service, arguing it was "extra work for a school project."

Resolution: I explained that monitoring was part of my assigned tasks and I needed metrics to create the dashboard. I offered to help implement the metrics by sharing example code. We eventually agreed, since our client (Gleb) also pushed us to do this.

Outcome: The query service metrics were added, and we actually found them useful for debugging performance issues during development.

9.7 Communication lessons learned

Overcommunicate rather than undercommunicate: Several integration issues happened because we made assumptions about what others were doing. Posting frequent updates in Discord prevented this.

Document decisions: When we made important decisions in meetings or chat, I started documenting them in a shared document. This created a reference when someone later asked "why did we do it this way?"

Be specific about needs: Saying "I need database access" is vague. Saying "I need the RDS endpoint, database name, and credentials added to the Kubernetes Secret in the rag-services namespace" gets faster results.

Acknowledge others' constraints: Not everyone could work at the same pace personal circumstances. Understanding this made coordination more flexible and reduced frustration.

10. Ethics & Sustainability Considerations

10.1 Overview

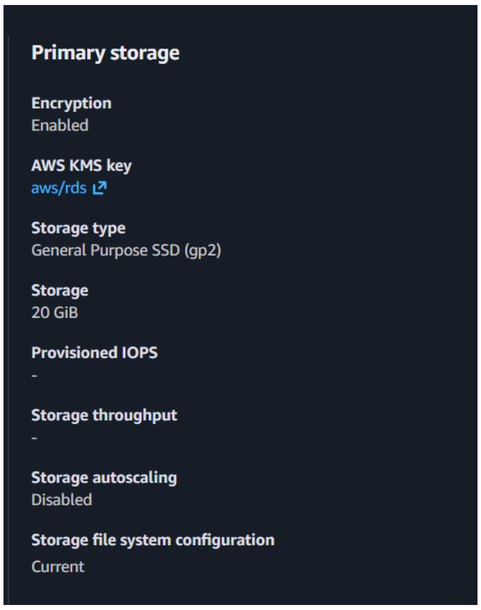
While implementing the RAG pipeline infrastructure, I considered ethical and sustainability aspects related to data handling, resource usage, and responsible computing. This section addresses how the implementation reflects these considerations in practical ways.

10.2 Data Privacy and Security

10.2.1 Embedding Data Protection

The RAG system processes and stores document embeddings, which are mathematical representations of text content. While embeddings are less readable than raw text, they still contain semantic information about the original documents.

Encryption at rest: All embeddings stored in PostgreSQL RDS are encrypted using AWS-managed encryption keys (AES-256). This is configured by default in RDS but I verified it was enabled. The S3 bucket storing document chunks also uses server-side encryption (SSE-S3). This ensures that even if storage media is physically compromised, the data remains protected.



Primary storage

Encryption
Enabled

AWS KMS key
aws/rds

Storage type
General Purpose SSD (gp2)

Storage
20 GiB

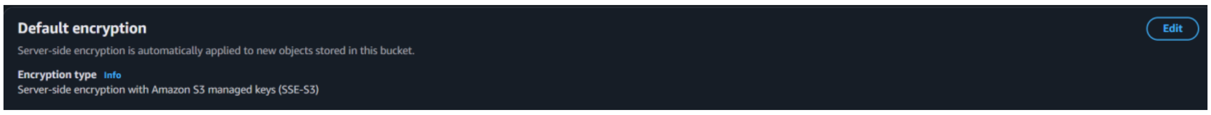
Provisioned IOPS
-

Storage throughput
-

Storage autoscaling
Disabled

Storage file system configuration
Current

This shows that the encryption is enabled for the RDS PostgreSQL Database



Default encryption

Server-side encryption is automatically applied to new objects stored in this bucket.

Encryption type Info
Server-side encryption with Amazon S3 managed keys (SSE-S3)

This shows that the encryption is enabled for the s3 bucket that stores the text chunks

Encryption in Transit: All communication between services uses HTTPS/TLS encryption. The PostgreSQL connection requires SSL (sslmode=require), preventing man-in-the-middle attacks. Communication between pods in Kubernetes uses the cluster network, which is isolated from the public internet.

Access Control: I implemented the principle of least privilege for database access. The embeddings-engine service account only has permissions to read from S3 and write to the embeddings table. It cannot access other tables or modify S3 bucket policies. This limits the damage if the service is compromised.

10.3 User data handling

The system processes documents uploaded by ADMINS only. I ensured the embeddings engine handles data responsibly:

No persistent logging of content: The service logs operational events (e.g., "processed 5 chunks") but doesn't log the actual text content. This prevents sensitive information from appearing in CloudWatch logs where it might be stored longer than necessary.

Automatic cleanup: Although not implemented in the current version, the S3 bucket has lifecycle policies configured to automatically delete old chunks after 90 days. This reduces the risk of old data accumulating unnecessarily.

10.4 Resource efficiency

10.4.1 Right-sizing infrastructure

Instance selection: I chose t3.medium instances for EKS nodes based on actual workload requirements rather than oversizing "just in case." The burstable nature of T3 instances means we only pay for extra CPU when needed, reducing costs during idle periods.

Resource limits: Each pod has memory limits (512Mi for embeddings-engine) that prevent resource hogging. If a bug causes a memory leak, Kubernetes kills the pod before it affects other services. This resource isolation improves overall system stability.

Scaling configuration: The node group scales between 1 and 4 nodes based on demand. During low usage periods (nights, weekends), Kubernetes can consolidate pods onto fewer nodes, allowing unused nodes to be terminated and saving costs.

The evidence of the aforementioned points have already been shown in the previous Chapters.

10.4.2 Storage optimization

Prometheus retention: I configured Prometheus to retain metrics for 30 days rather than indefinitely. This balances the need for historical data with storage costs. Longer retention would require additional storage volumes and backup costs.

ECR image lifecycle: Container images in ECR use lifecycle policies that keep only the last 10 versions. Old images are automatically deleted, preventing storage costs from growing over time as we push new versions.

10.5 Cost awareness

10.5.1 Monthly cost estimate

Understanding the cost impact of infrastructure decisions helps make responsible choices:

- **EKS Control Plane:** $72/month (fixed cost)
- **3x t3.medium nodes:** ~$90/month (can scale down to 1 node = $30/month)
- **RDS db.t3.medium:** ~$50/month
- **Qdrant EC2 t3.small:** ~$15/month
- **Data Transfer:** ~$5-10/month
- **S3 Storage:** ~$2/month (minimal)
- **Total:** ~$234-244/month at full scale

10.5.2 Cost Optimization Decisions:

Using SPOT Instances: Although not implemented in the current configuration, EKS nodes could use SPOT instances for 60-70% cost savings. I documented this as a future improvement because SPOT instances can be terminated by AWS with short notice, which requires additional handling logic.

Internal Load Balancers Only: Grafana uses an internal load balancer instead of a public one. Internal load balancers are cheaper and align with security requirements.

Single Availability Zone for Dev: For development, running in a single AZ would reduce costs by ~30% (no cross-AZ data transfer, fewer redundant resources). Production would require multi-AZ for reliability.

10.6 Sustainable Computing Practices

10.6.1 Energy Efficiency

Serverless where possible: Although the embeddings engine runs as always-on pods, future iterations could use AWS Lambda for sporadic workloads. Lambda only consumes resources when processing requests, reducing idle energy consumption.

Efficient container images: The Docker images use slim base images (python:3.11-slim) rather than full OS images. Smaller images reduce storage, network transfer, and startup time, all of which contribute to lower energy usage.

10.6.2 Monitoring for waste prevention

Resource utilization tracking: The Grafana dashboards track CPU and memory usage, making it visible when resources are underutilized. If nodes consistently run at 20% CPU, that indicates we could downsize and reduce waste.

Idle resource detection: Monitoring alerts (though not fully implemented) could notify when services are idle but still consuming resources. This awareness enables better resource management decisions.

10.7 Responsible AI Considerations

10.7.1 Model Choice and Efficiency

Using Portkey API cohere embedding model: We use a cohere embedding model provided by our teacher/client via Portkey API. We could consider using a different model, such as a smaller model, to reduce costs and computational power. It's faster and since it uses less resources it's better for the environment, compared to a larger model.

Batch processing: The /process/s3 endpoint processes multiple chunks in a batch rather than making individual API calls for each chunk. This reduces API overhead and network traffic, making the system more efficient.

10.7.2 Avoiding overprocessing

Embeddings are cached: Once a document chunk is embedded and stored, it's never reprocessed unless explicitly updated. This prevents redundant computation and API calls. The dual-storage architecture ensures embeddings survive system failures without needing regeneration.

10.8 Ethical Considerations in System Design

10.8.1 Transparency

Health checks: The service exposes its operational status through the /health endpoint. Users and operators can verify that the system is functioning correctly, including database connectivity. This transparency helps build trust.

Logging: The service logs important operational events without exposing sensitive data. This balance allows debugging and monitoring while respecting privacy.

10.9 Graceful Failure Handling

Partial degradation: If Qdrant becomes unavailable, the system continues functioning with PostgreSQL. This graceful degradation means temporary infrastructure issues don't completely break the service.

Error messages: When the service fails to process a chunk, it logs the error and continues with remaining chunks rather than failing the entire batch. This resilient behavior prevents cascading failures.

10.10 Future Improvements for Ethics and Sustainability

Automated cost alerts: Set up CloudWatch alarms that trigger when monthly costs exceed expected thresholds. This prevents unexpected cost overruns from bugs or misconfigurations.

Data retention policies: Implement automatic deletion of embeddings after a defined period (e.g., 1 year) unless explicitly marked for long-term retention. This reduces storage costs and privacy risks.

Audit logging: Add audit trails for who accessed which documents and when. This accountability improves security and enables compliance with data protection regulations.

11. Personal Reflection & Learning

11.1 Personal Learning Goals

At the start of this proftaak, I set several learning goals focused on cloud infrastructure and Kubernetes:

Goal 1: Learn Kubernetes in production context - I wanted to move beyond tutorial-level Kubernetes knowledge and understand how to run production workloads on EKS. This included setting up proper IAM roles, handling secrets, and configuring resource limits.

Goal 2: Implement monitoring from scratch - In previous projects, monitoring was either pre-configured or skipped entirely. I wanted to understand how to set up a complete observability stack and create custom dashboards for application-specific metrics.

Goal 3: Understand database integration patterns - I had worked with databases before but never integrated multiple storage systems or handled external databases from Kubernetes. I wanted to learn how services discover and connect to databases securely.

Goal 4: Practice Infrastructure as Code - While I had used Terraform in CS1, I wanted to deepen my understanding by managing more complex resources like EKS clusters, Helm releases, and Kubernetes manifests through Terraform.

11.2 What went well

EKS cluster setup: The EKS cluster deployment went smoother than expected. The Terraform AWS provider documentation was clear, and following the official patterns for IAM roles and OIDC providers worked on the first try. The modular structure I used (separate files for cluster, IAM, security groups) made debugging easier when I needed to adjust configurations.

IRSA implementation: Setting up IAM Roles for Service Accounts was surprisingly straightforward once I understood the trust relationship between the OIDC provider and IAM. This was a significant learning moment because it showed me how Kubernetes and AWS IAM can integrate securely without hardcoding credentials.

Monitoring stack deployment: Using the Helm chart for Prometheus and Grafana saved enormous time compared to manual installation. The pre-built dashboards gave immediate value, and learning how ServiceMonitors work helped me understand Kubernetes custom resources better.

Team collaboration: Working with teammates who handled other services (chunking, query) taught me how to design loosely coupled systems. We agreed on API contracts early, which meant I could develop the embeddings engine independently without constant coordination.

11.3 Challenges and Solutions

11.3.1 Challenge 1: Qdrant External Service Integration

Problem: Qdrant runs on an EC2 instance outside the cluster, and I needed Prometheus to scrape metrics from it. Initially, I tried using an ExternalName service, but Prometheus couldn't discover it because ServiceMonitors only work with Endpoints.

Solution: I researched and found the pattern of creating a headless Service with manually defined Endpoints pointing to the external IP. This "bridged" the external service into Kubernetes service discovery. I learned this from a Stack Overflow post and Prometheus documentation on monitoring external targets.

Learning: Kubernetes assumes everything runs as pods. When integrating external systems, you need creative solutions like endpoint bridges or sidecar exporters. This pattern will be useful in future projects with hybrid cloud/on-premises setups.

11.3.2 Challenge 2: PostgreSQL Initialization Timing

Problem: The embeddings-engine pods sometimes crashed on startup because they tried to create the vector extension before having proper permissions. The RDS instance exists, but the extension wasn't installed yet.

Solution: I modified the initialization code to gracefully handle failures and print warnings instead of crashing. I also ensured the RDS instance had the vector extension pre-installed through Terraform. This made the service more resilient to temporary database unavailability.

Learning: Services should fail gracefully and retry rather than crash immediately. This defensive programming makes systems more reliable in production where dependencies might be temporarily unavailable.

11.3.3 Challenge 3: Understanding Prometheus Queries

Problem: Creating the custom dashboard required writing PromQL queries, which I had never done before. The syntax for calculating average latency using rate() and division wasn't intuitive.

Solution: I spent time in the Prometheus query browser experimenting with different queries and reading the PromQL documentation. I also looked at examples from existing dashboards to understand common patterns. Eventually, I understood how rate() converts counters to per-second rates and how to combine metrics.

Learning: Prometheus query language is powerful but requires practice. Understanding counters vs gauges and when to use rate() vs increase() is crucial for accurate metrics. This skill will be valuable in any monitoring context.

11.3.4 Challenge 4: Security Group Configuration

Problem: Initially, my security group rules were too permissive. I had opened all ports between the EKS nodes and everything else "to make it work." This isn't production-ready.

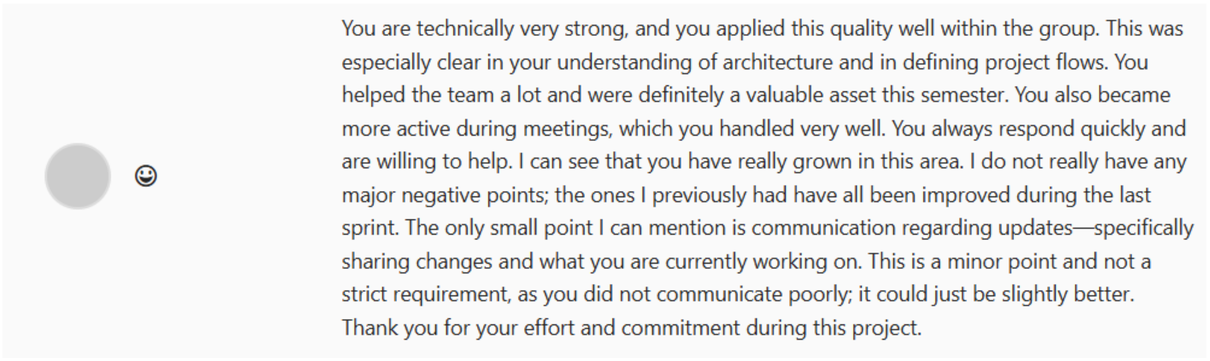
Solution: I went through each security group systematically and identified which specific ports each service actually needs. I tested connectivity after each change to ensure nothing broke. The final configuration follows the principle of least privilege.

Learning: Security should be designed in from the start, not added later. It's tempting to open everything during development, but cleaning it up afterwards is tedious. Next time, I'll start with restrictive rules and only open what's necessary.

11.4 Feedback Received and Response

A crucial part of this project was learning how to work effectively in a team. Below is the specific feedback I received from my peers during the final assessment, along with my reflections and the actions I took to improve.

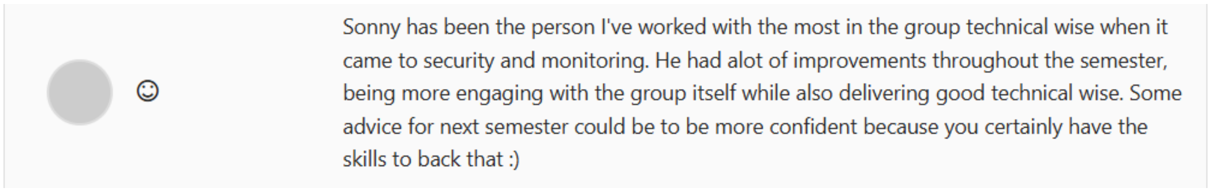
11.4.1 Feedback on Architecture and Communication


☺

You are technically very strong, and you applied this quality well within the group. This was especially clear in your understanding of architecture and in defining project flows. You helped the team a lot and were definitely a valuable asset this semester. You also became more active during meetings, which you handled very well. You always respond quickly and are willing to help. I can see that you have really grown in this area. I do not really have any major negative points; the ones I previously had have all been improved during the last sprint. The only small point I can mention is communication regarding updates—specifically sharing changes and what you are currently working on. This is a minor point and not a strict requirement, as you did not communicate poorly; it could just be slightly better. Thank you for your effort and commitment during this project.

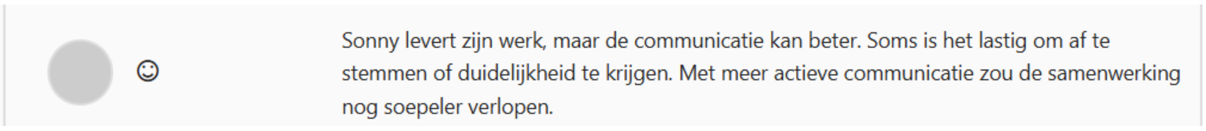
- **The feedback:** This peer highlighted my technical strength in defining the architecture and project flows. However, they constructively pointed out that my communication regarding updates, specifically sharing *what* I was working on at any given moment, could be improved.
- **Reflection & action:** It was validating to hear that my architectural decisions helped the team, as I spent a lot of time designing the EKS setup. However, the point about updates was spot on. I tend to "tunnel vision" when I code.
- **How I will improve:** To fix this, I will stop waiting for major milestones to share progress. I should start messaging in Whatsapp about what I have finished and what I'm working on.

11.4.2 Feedback on Confidence (Security & Monitoring)


Sonny has been the person I've worked with the most in the group technical wise when it came to security and monitoring. He had alot of improvements throughout the semester, being more engaging with the group itself while also delivering good technical wise. Some advice for next semester could be to be more confident because you certainly have the skills to back that :)

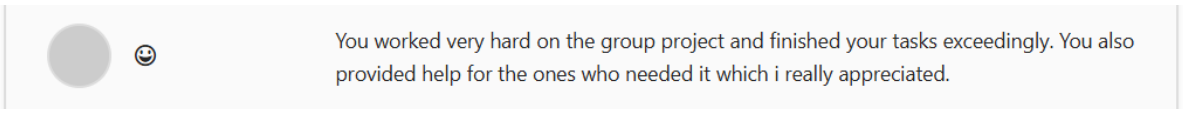
- **The Feedback:** This came from the teammate I worked with most closely on the security and monitoring stack. They noted my improvement in engagement but advised me to be more confident because I "have the skills to back that".
- **Reflection & Action:** This really stuck with me. In the first half of the semester, I often held back my opinions during discussions because I didn't want to overstep.
- **How I will improve:** In future projects, I will try to take the lead for often when it comes to certain decisions, be it functional or technical decisions, instead of just suggesting it.

11.4.3 Feedback on alignment


Sonny levert zijn werk, maar de communicatie kan beter. Soms is het lastig om af te stemmen of duidelijkheid te krijgen. Met meer actieve communicatie zou de samenwerking nog soepeler verlopen.

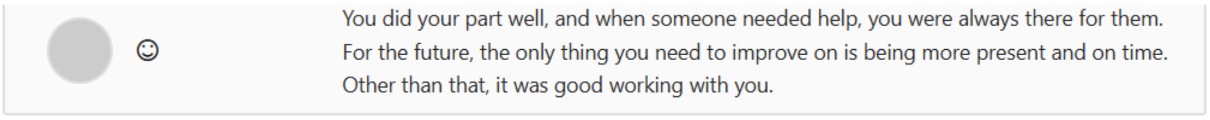
- **The Feedback:** This peer noted that while I deliver good work, communication could be better, stating that "sometimes it is difficult to align or get clarity".
- **Reflection & Action:** This was the toughest feedback to hear, but arguably the most important. "Delivering work" isn't enough if my teammates aren't up to date with it. This might cause stress or uncertainty, since they wouldn't know if I finished the tasks or not (which I did)
- **How I will improve:** I realized that *my* clarity isn't the same as *team* clarity. I should in the future explicitly tell and explain my teammates on what part I have finished and what part I have worked on.

11.4.4 Feedback on Helpfulness


You worked very hard on the group project and finished your tasks exceedingly. You also provided help for the ones who needed it which i really appreciated.

- **The Feedback:** This peer appreciated that I finished my tasks exceedingly well and, more importantly, provided help to those who needed it.
- **Reflection:** I am very proud of this. It’s nice to hear that I have done my tasks really well and that I have been helpful.
- **How I maintained this:** I made it a priority to jump into calls when the frontend team was struggling with CORS issues or database connectivity, treating their blockers as my own.

11.4.5 1.4.5 Feedback on Punctuality


You did your part well, and when someone needed help, you were always there for them. For the future, the only thing you need to improve on is being more present and on time. Other than that, it was good working with you.

- **The Feedback:** While they noted I was "always there" to help, this peer pointed out that I need to improve on being present and on time.
- **Reflection & Action:** I realized that being late to stand-ups, even by just 5 minutes, disrespects the team's time and kills the momentum of the meeting.
- **How I will improve:** This is a simple discipline fix. I will set calendar notifications 15 minutes before every meeting to ensure I was settled and ready to go before the call started, rather than rushing in at the last minute.

11.5 Teacher feedback

11.5.1 Feedback:

- "Your technical sections are detailed, but you need to show more depth in research. Why did you choose these solutions?"
- "Add reflection on what you learned and how you grew from challenges."

I added a section with the DOT framework research and comparison tables. I expanded my personal reflection to include specific learning moments and challenges. This feedback significantly improved the documentation quality.

11.6 What I would do differentky

Better planning for dependencies: I started implementing the embeddings engine before fully understanding how it would be triggered by the chunking service. This caused some confusion about the S3 event flow. Next time, I would diagram the complete data flow before writing code.

Earlier testing: I deployed everything to EKS and only then tested if the databases were reachable. This made debugging harder because multiple things could be wrong. I should have tested database connectivity from a simple test pod first, then added the application logic.

Documentation as I go: I wrote most of this documentation after finishing the implementation. This meant I had to reconstruct why I made certain decisions. Keeping a log of decisions and research while working would make documentation much easier.

Resource limits from day 1: I initially deployed pods without resource limits, which caused issues when one pod consumed too much memory and got killed. Setting proper requests and limits from the start would have avoided this problem.

11.7 Achievement of learning goals

Goal 1 (Kubernetes Production): Achieved. I now understand how to properly configure EKS with IAM roles, security groups, and resource management. I can deploy services with health checks and handle secrets securely.

Goal 2 (Monitoring Implementation): Achieved. I successfully deployed Prometheus and Grafana, created custom dashboards, and understand how ServiceMonitors enable automatic discovery. I can troubleshoot issues using metrics.

Goal 3 (Database integration): Partially achieved. I successfully connected services to PostgreSQL and Qdrant, but I didn't have to handle complex scenarios like connection pooling or failover. This is something I want to learn more about.

Goal 4 (Infrastructure as Code): Achieved. I'm comfortable with Terraform for managing AWS resources and Kubernetes resources together. I understand state management and how to structure modules for reusability.

11.8 Skills Gained

Technical Skills:

- EKS cluster management and configuration
- Helm chart deployment and customization
- Prometheus and Grafana for Kubernetes
- ServiceMonitor and custom metrics
- IAM Roles for Service Accounts (IRSA)
- Security group design and troubleshooting
- PromQL query language

Soft Skills:

- Breaking down complex problems into manageable tasks
- Reading documentation effectively
- Debugging distributed systems
- Communicating technical decisions to teammates
- Time management for parallel development

11.9 Areas for future improvement

Alerting: I focused on metrics collection and visualization but didn't set up proper alerting rules in Alertmanager. Learning how to define meaningful alerts and integrate with notification systems like Slack would be valuable.

High availability: The current setup runs services with 2 replicas but doesn't test failure scenarios. I want to learn more about pod disruption budgets, graceful shutdowns, and handling node failures.

Cost optimization: I used the instance types recommended in documentation but didn't deeply analyze actual resource usage to optimize costs. Learning cost analysis tools and right-sizing techniques would be useful.

11.10 Reflection

At the start of this proftaak, I approached infrastructure mainly as a technical execution problem: get the cluster running, make services communicate, and fix issues as they appear. Over the course of the project, my perspective shifted toward viewing infrastructure as a system of long-term decisions, where clarity, communication, and resilience matter as much as correctness.

I noticed a recurring pattern in my challenges: most issues were not caused by lack of technical knowledge, but by incomplete upfront reasoning or assumptions left unvalidated. Examples include database initialization timing, security group permissiveness, and late testing of dependencies. This taught me that in production-like environments, defensive design and early validation are more important than fast implementation.

On a personal level, the feedback about communication and alignment highlighted that technical output alone is insufficient in a team setting. I learned that engineering responsibility includes making work visible, not just correct. As a result, I now consciously plan communication moments alongside technical milestones.

Overall, this project moved me from a “builder” mindset toward a more professional infrastructure engineer mindset, where reliability, clarity, and collaboration are first-class concerns.