**ECE – 573 PROJECT – 4**

**SECTION II**

**Cluster.yml**

kind: Cluster

apiVersion: kind.x-k8s.io/v1alpha4

nodes:

- role: control-plane

- role: worker

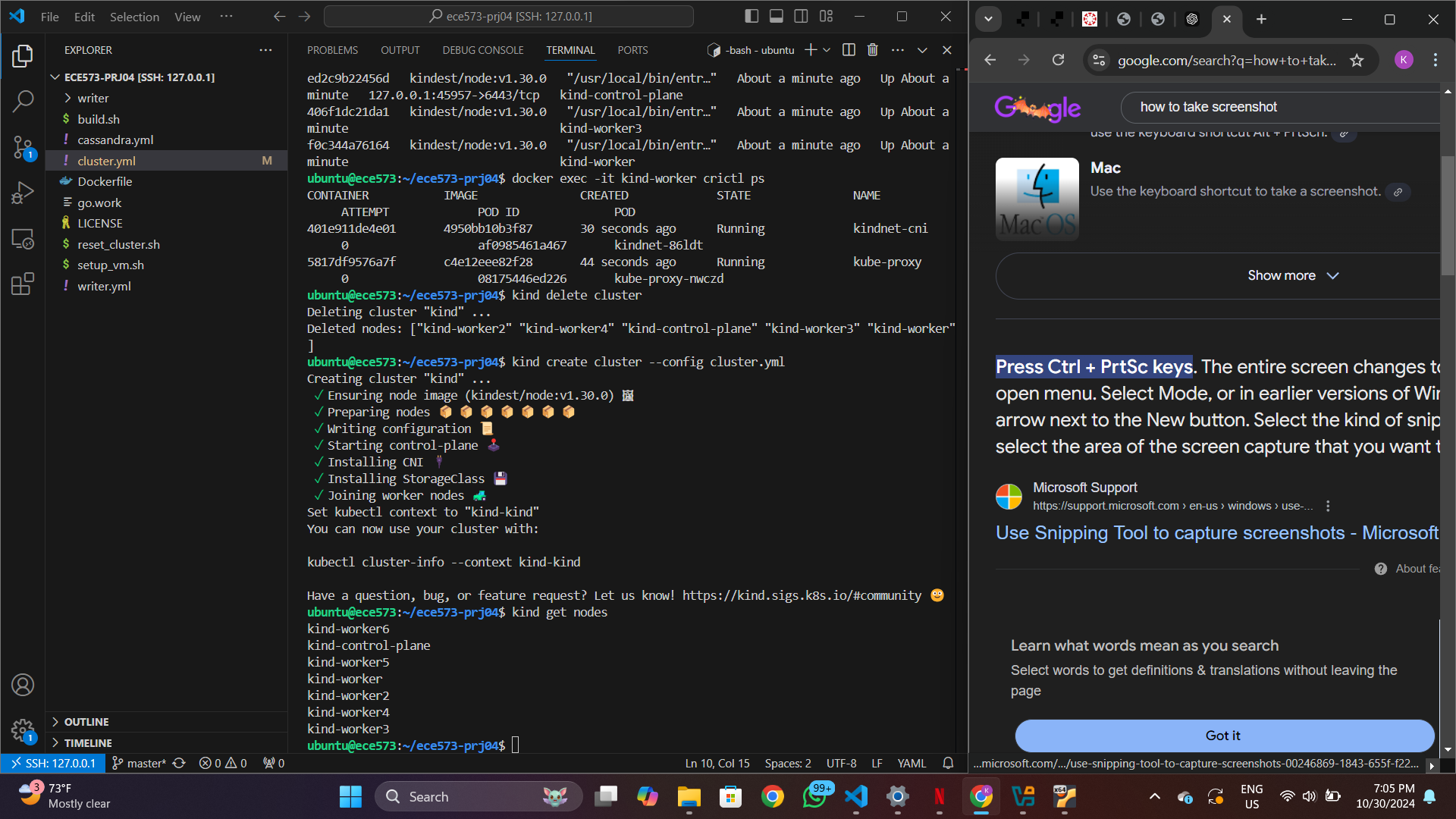
- role: worker

- role: worker

- role: worker

- role: worker

- role: worker



**SECTION III**

**Cassandra.yml**

apiVersion: v1

kind: Service

metadata:

  name: cassandra-service

spec:

  clusterIP: None

  ports:

  - port: 9042

  selector:

    app: cassandra

---

apiVersion: apps/v1

kind: StatefulSet

metadata:

  name: cassandra

spec:

  serviceName: cassandra-service

  replicas: 5   Change this value to 5

  selector:

    matchLabels:

      app: cassandra

  template:

    metadata:

      labels:

        app: cassandra

    spec:

      containers:

      - name: cassandra

        image: cassandra:4.1.3

        ports:

        - containerPort: 9042

        resources:

          limits:

            cpu: "500m"

            memory: 1Gi

          requests:

            cpu: "500m"

            memory: 1Gi

        env:

          - name: MAX\_HEAP\_SIZE

            value: 512M

          - name: HEAP\_NEWSIZE

            value: 128M

          - name: CASSANDRA\_SEEDS

            value: "cassandra-0.cassandra-service.default.svc.cluster.local"

          - name: CASSANDRA\_CLUSTER\_NAME

            value: "ece573-prj04"

        volumeMounts:

        - name: cassandra-data

          mountPath: /var/lib/cassandra

  volumeClaimTemplates:

  - metadata:

      name: cassandra-data

    spec:

      accessModes: [ "ReadWriteOnce" ]

      resources:

        requests:

          storage: 10Gi

A computer screen shot of a computer screen

Description automatically generated

**1. Explain the "resources" section for the Cassandra container from `cassandra.yml`. What are the differences between "limits" and "requests"?**

In Kubernetes, the `resources` section for a container allows you to define how much CPU and memory the container can request and use. The two primary fields within `resources` are `limits` and `requests`. The \*\*requests\*\* field specifies the minimum amount of CPU and memory that the container requires. Kubernetes uses these values to ensure that the container is scheduled onto a node that has at least this amount of resources available. This guarantees the container can function reliably at this minimum resource level. Meanwhile, \*\*limits\*\* represent the maximum CPU and memory the container is allowed to consume. If the container attempts to exceed these limits, it will be throttled for CPU or, in the case of memory, could even be terminated if it uses more than what is specified. In this `cassandra.yml`, both `limits` and `requests` are set to the same values (`cpu: "500m"` and `memory: 1Gi"`), ensuring the container receives exactly these resources and cannot go above them.

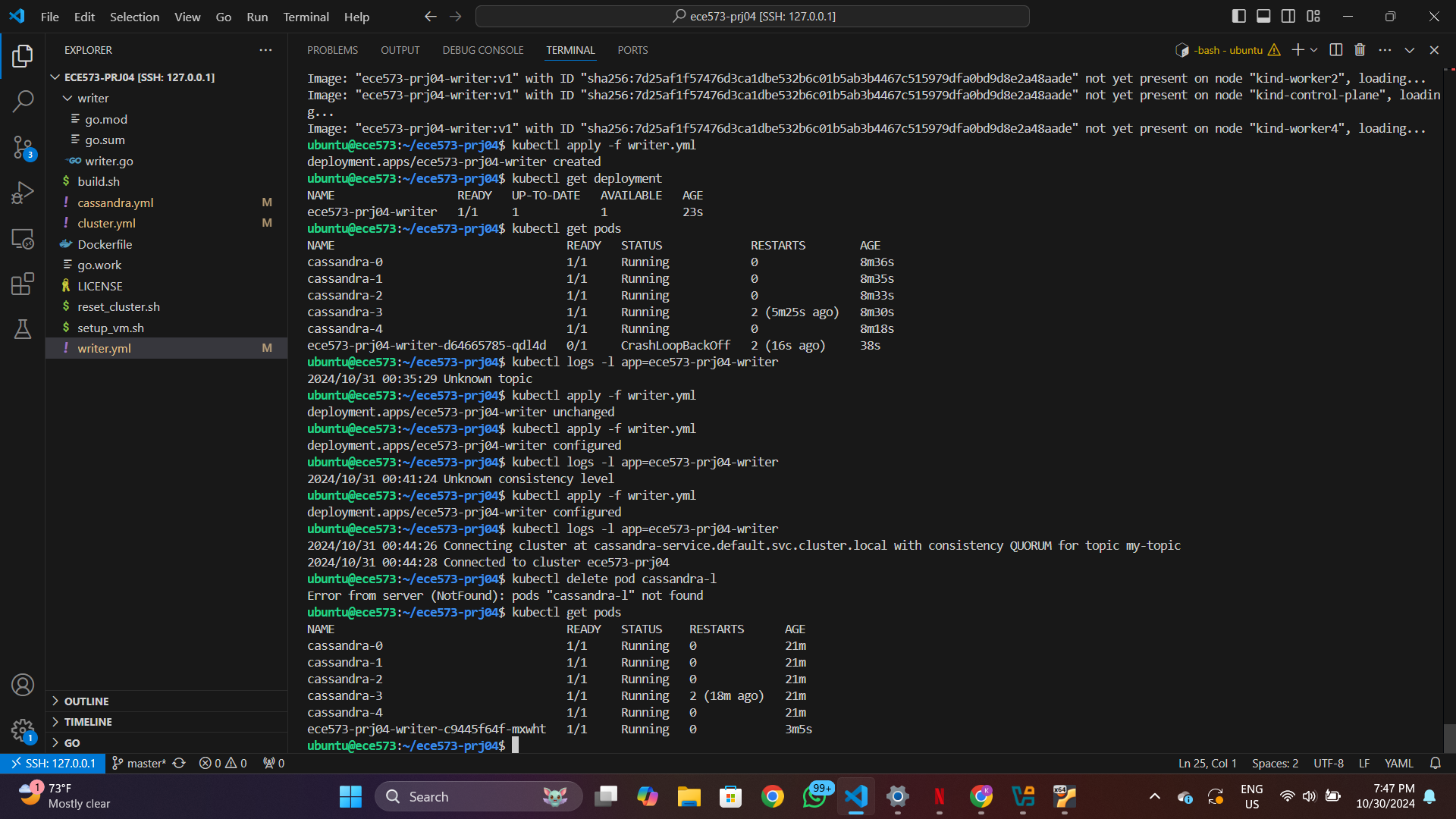
**2. Below the "resources" section, you find the "env" section to set up environment variables. Where was the corresponding part for Project 3? Explain the difference between the two.**

The `env` section in `cassandra.yml` defines environment variables directly within the Kubernetes configuration for the Cassandra container, allowing customization of the Cassandra service at runtime. In Project 3, the environment variables were likely set up within the Go client programs (`writer.go` and `reader.go`) to configure them to connect to the Cassandra cluster. These could be specified in the application code, passed through a configuration file, or possibly set up as command-line arguments, depending on the needs of the client programs. The main difference lies in the scope and purpose of the variables: \*\*the `env` section in Kubernetes\*\* defines settings that directly affect the configuration and behavior of the Cassandra container itself, such as memory limits (`MAX\_HEAP\_SIZE`), the cluster’s name (`CASSANDRA\_CLUSTER\_NAME`), and the initial seed node for discovery (`CASSANDRA\_SEEDS`). In contrast, \*\*Project 3’s environment variables\*\* are specific to the client programs, focusing on details like connection parameters to the Cassandra database and possibly settings around data writing and reading operations.

**3. How does the Cassandra service know which Pods are part of the service?**

The Cassandra service determines which pods belong to it by using a \*\*selector\*\* that matches specific labels on the pods. In the `cassandra.yml` file, the service specification includes a selector field with the label `app: cassandra`. This label is also applied to each Cassandra pod within the StatefulSet configuration under `template -> metadata -> labels`. When Kubernetes creates a pod with a matching label, it automatically becomes part of the service. Since this particular service is configured as a \*\*headless service\*\* (with `clusterIP: None`), it doesn’t provide a single IP address as a standard service would. Instead, it assigns each pod in the StatefulSet its own DNS entry, enabling the Cassandra nodes to discover each other directly and form a cluster. This DNS-based pod discovery is essential for Cassandra’s internal mechanisms for peer-to-peer communication and data replication.

**SECTION IV**

****

1. How does the writer deployment connect to the Cassandra service? In particular, where does "cassandra-service.default.svc.cluster.local" come from?

The writer deployment connects to the Cassandra service through the `CASSANDRA\_SEEDS` environment variable, which is set to `"cassandra-service.default.svc.cluster.local"`. This address is the fully qualified domain name (FQDN) of the Cassandra service within the Kubernetes cluster. The structure of this FQDN is determined by Kubernetes' internal DNS system, which automatically resolves service names to the corresponding pods. Here, `"cassandra-service"` is the name of the service as defined in `cassandra.yml`. The `"default"` segment represents the Kubernetes namespace in which both the service and writer deployment are running (usually `default` unless specified otherwise), and `"svc.cluster.local"` is the cluster-wide domain suffix managed by Kubernetes DNS. Together, this address allows the writer deployment to locate and connect to the Cassandra pods through the headless service, enabling communication between the client (writer) and the database nodes.

2. We add retrying logic to the writer program in Project 3. Do we need it for Project 4?

Yes, retrying logic is still beneficial for Project 4. In distributed environments, intermittent connectivity issues, network delays, or temporary unavailability of Cassandra nodes can still occur, even if the underlying Kubernetes setup helps with load balancing and fault tolerance. The retrying logic ensures that if a request fails due to a transient issue, the writer program will attempt to reconnect rather than failing immediately. This increases the resilience of the application and helps prevent data loss or inconsistency, especially during high load or when nodes may briefly become unavailable. Hence, implementing retry logic in Project 4 can help maintain a stable and reliable connection to the Cassandra service, similar to its role in Project 3.

writer.go

package main

import (

    "log"

    "math/rand"

    "os"

    "strings"

    "github.com/gocql/gocql"

)

func main() {

    topic := os.Getenv("TOPIC")

    if topic == "" {

        log.Fatalf("Unknown topic")

    }

    cs := os.Getenv("CONSISTENCY")

    consistency := gocql.All

    switch strings.ToUpper(cs) {

    case "ALL":

    case "ONE":

        consistency = gocql.One

    case "QUORUM":

        consistency = gocql.Quorum

    default:

        log.Fatalf("Unknown consistency level %s", cs)

    }

    seed := os.Getenv("CASSANDRA\_SEEDS")

    log.Printf(

        "Connecting cluster at %s with consistency %s for topic %s",

        seed, consistency, topic)

    cluster := gocql.NewCluster(seed)

    cluster.Consistency = consistency

    session, err := cluster.CreateSession()

    if err != nil {

        log.Fatalf("Cannot connect to cluster at %s: %v", seed, err)

    }

    defer session.Close()

    var clusterName string

    if err := session.Query(

        "SELECT cluster\_name FROM system.local").

        Scan(&clusterName); err != nil {

        log.Fatalf("Cannot query cluster: %v", err)

    }

    log.Printf("Connected to cluster %s", clusterName)

    if err := session.Query(

        `CREATE KEYSPACE IF NOT EXISTS ece573

            WITH replication = {

                'class':'SimpleStrategy',

                'replication\_factor':3}`).

        Exec(); err != nil {

        log.Fatalf("Cannot create keyspace ece573: %v", err)

    }

    if err := session.Query(

        `CREATE TABLE IF NOT EXISTS ece573.prj04 (

            topic text, seq int, value double,

            PRIMARY KEY (topic, seq))`).

        Exec(); err != nil {

        log.Fatalf("Cannot create table ece573.prj04: %v", err)

    }

    if err := session.Query(

        `CREATE TABLE IF NOT EXISTS ece573.prj04\_last\_seq (

            topic text, seq int,

            PRIMARY KEY (topic))`).

        Exec(); err != nil {

        log.Fatalf("Cannot create table ece573.prj04\_last\_seq: %v", err)

    }

    log.Printf("Tables ece573.prj04 and ece573.prj04\_last\_seq ready.")

    // Read lastSeq from ece573.prj04\_last\_seq

    var lastSeq int

    if err := session.Query(`SELECT seq FROM ece573.prj04\_last\_seq WHERE topic = ?`, topic).Scan(&lastSeq); err != nil {

        if err == gocql.ErrNotFound {

            lastSeq = 0 // If not found, start from 0

        } else {

            log.Fatalf("Cannot read lastSeq from ece573.prj04\_last\_seq: %v", err)

        }

    }

    log.Printf("%s: start from lastSeq=%d", topic, lastSeq)

    for seq := lastSeq + 1; ; seq++ {

        value := rand.Float64()

        log.Printf("Inserting %d with value %f", seq, value)

        err := session.Query(

            `INSERT INTO ece573.prj04 (topic, seq, value) VALUES (?, ?, ?)`,

            topic, seq, value).

            Exec()

        if err != nil {

            log.Fatalf("Cannot write %d to table ece573.prj04: %v", seq, err)

        }

        err = session.Query(

            `INSERT INTO ece573.prj04\_last\_seq (topic, seq) VALUES (?, ?)`,

            topic, seq).

            Exec()

        if err != nil {

            log.Fatalf("Cannot write %d to table ece573.prj04\_last\_seq: %v", seq, err)

        }

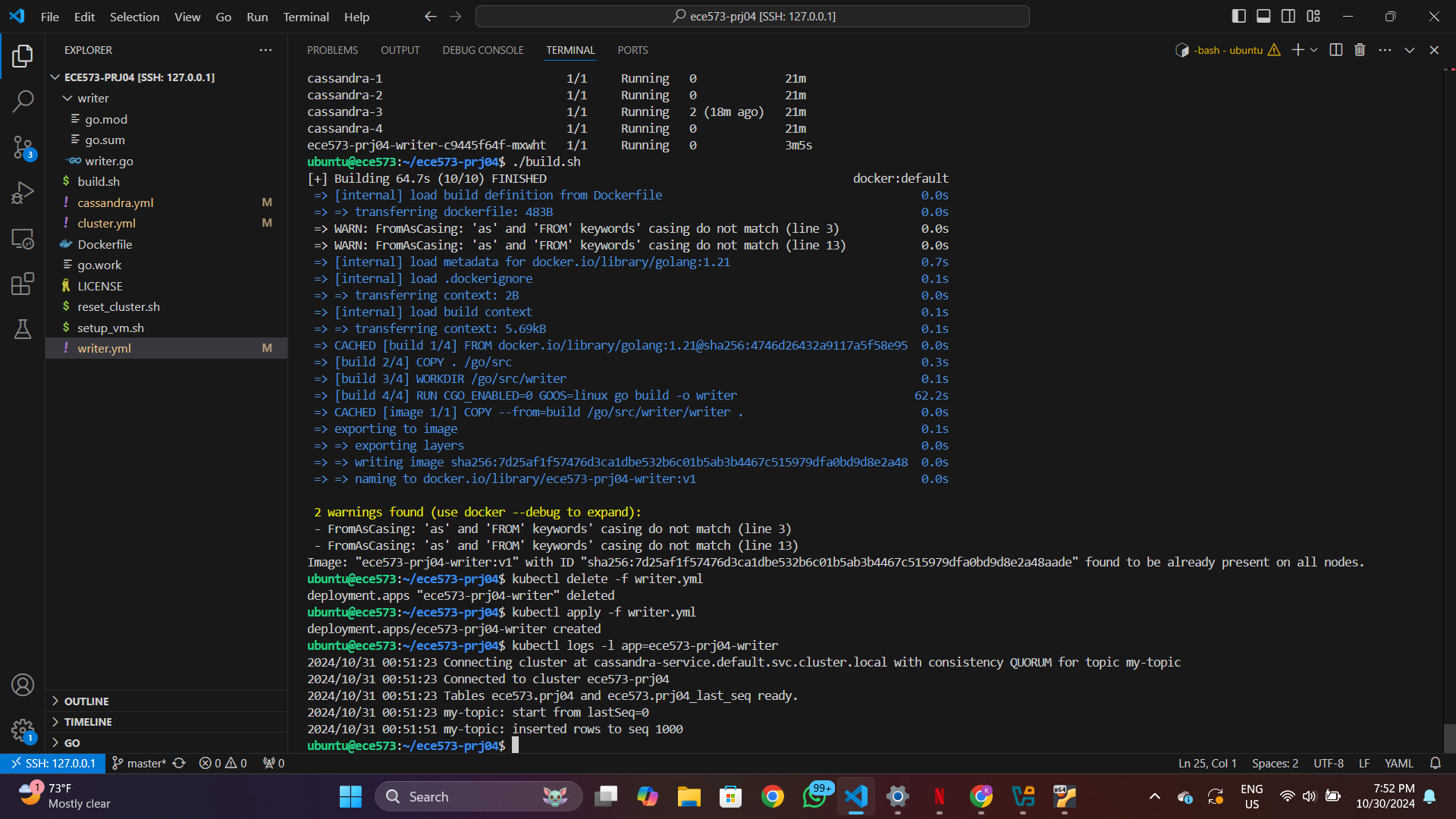
        if seq%1000 == 0 {

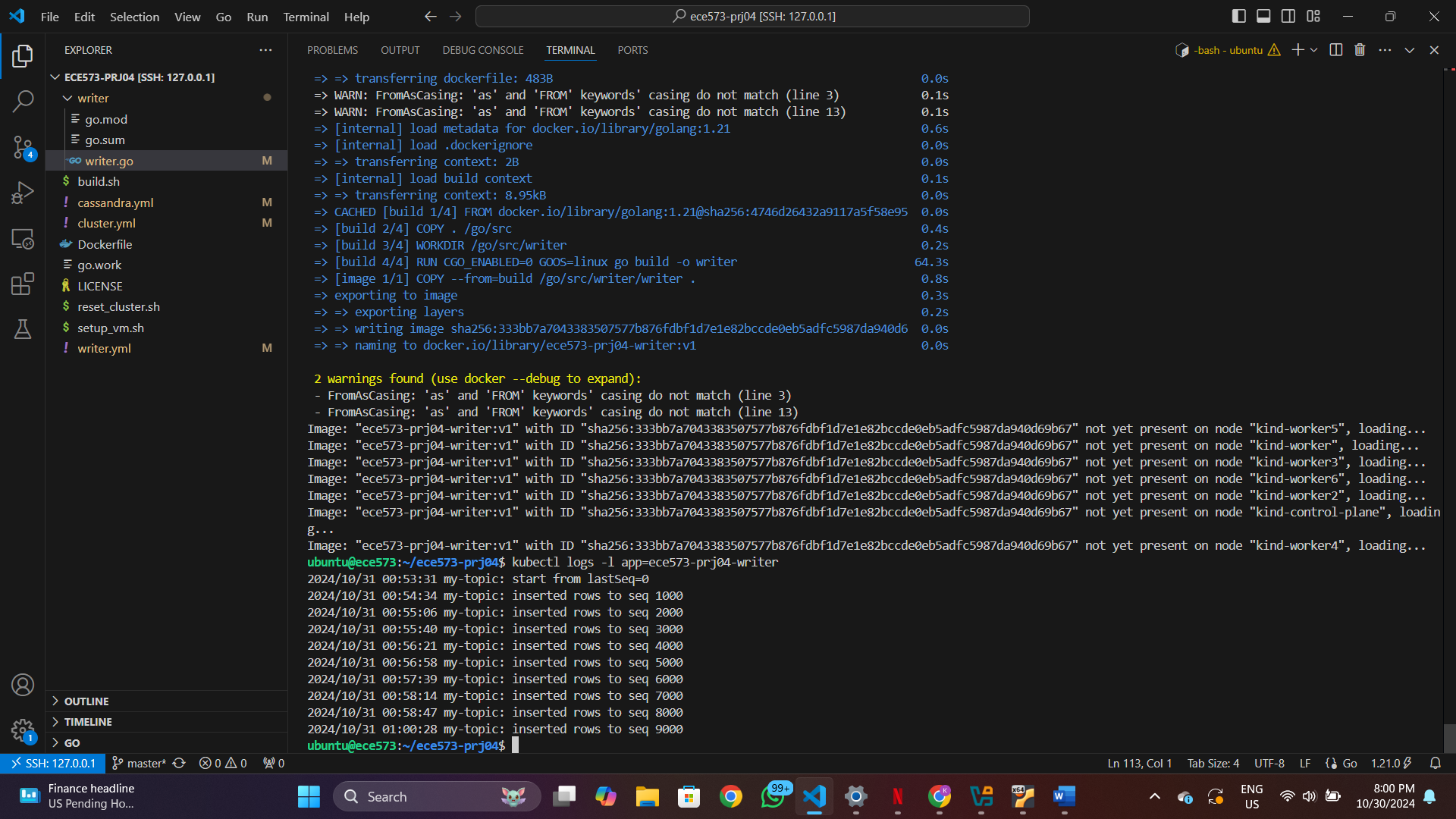
            log.Printf("%s: inserted rows to seq %d", topic, seq)

        }

    }

}





What happens if consistency is broken so that the writer doesn't obtain the most recent last seq? Will this cause an issue for the writer?

If consistency is broken and the writer doesn’t obtain the most recent `seq` (sequence number), it can lead to issues like \*\*out-of-order writes\*\* and \*\*data overwrites\*\*. Without the latest sequence number, the writer may operate on outdated information, unaware of recent changes in the database. This can result in it writing data based on a stale state, potentially overwriting more recent updates. Such inconsistencies can be especially problematic in applications that rely on strict ordering or cumulative accuracy, as it may cause gaps or incorrect ordering in the data history. In distributed systems like Cassandra, consistency problems can lead to these kinds of data integrity issues if the writer does not have mechanisms (such as retries or version checks) to ensure it’s working with the latest state.