Go Developer Technical Challenge - 2

Technical Challenge: Develop a Kubernetes Operator to Simulate a Distributed Job Scheduling System

Objective: Create a Kubernetes Operator using Go that simulates a distributed job scheduling system. The operator should manage compute nodes as Kubernetes Custom resource definition and handle the lifecycle of nodes and jobs within the system.

Requirements

1. **Custom Resource Definitions (CRDs)**
   - Define two CRDs: `ComputeNode` and `ComputeJob`.
     - `ComputeNode` CRD:
       - `spec.resources`: The resources available to the node (e.g., CPU, memory).
       - `status.state`: The current state of the node (e.g., `Pending`, `Running`, `Failed`).
     - `ComputeJob` CRD:
       - `spec.command`: The command to run as a job.
       - `spec.nodeSelector`: Criteria for selecting nodes to run the job.
       - `spec.parallelism`: The number of nodes the job should run on simultaneously.
       - `status.state`: The current state of the job (e.g., `Pending`, `Running`, `Completed`, `Failed`).
       - `status.startTime`: The start time of the job.
       - `status.endTime`: The end time of the job.
       - `status.activeNodes`: The list of nodes where the job is currently running.
2. **Operator Functionality**
   - Implement controllers for `ComputeNode` and `ComputeJob` resources (you can also improvise with any resources that you judge useful to achieve the purpose of the assessment) .
   - The `ComputeNode` resource represents an abstraction of the physical node.
   - Create a `ComputeNode` for each physical node.
   - Manage the lifecycle of `ComputeNode` resources, ensuring proper state transitions (e.g., from `Pending` to `Running`).
   - When a new `ComputeJob` is created, schedule it on the appropriate number of `ComputeNode` instances based on `spec.nodeSelector` and `spec.parallelism`.
   - Ensure the job runs as Kubernetes pods on the selected `ComputeNode` instances.
   - Update the `ComputeJob` status with the job's state, start time, end time, and active nodes.
   - Handle updates to `ComputeJob` resources, ensuring the running job is updated accordingly.
   - Handle deletions of `ComputeJob` resources, ensuring the associated pods are also deleted.
   - Manage the relationship between `ComputeNode` and `ComputeJob`, ensuring jobs are properly scheduled and run on available nodes.
3. **Error Handling & Logging**
   - Implement robust error handling to manage failures in creating/updating/deleting resources.
   - Include detailed logging for all significant operations and state changes.
4. **Testing**
   - Write unit tests for the controller logic.
   - Provide end-to-end tests to verify the operator's functionality in a real Kubernetes cluster.
   - Ensure tests cover scenarios such as successful job scheduling, node state transitions, job parallelism, and handling update conflicts.

Submission Requirements

1. **Code Submission**

   - Provide the complete source code for the Kubernetes Operator, including the CRD definitions, controller implementations, and any necessary auxiliary files (e.g., Dockerfile, Makefile).

2. **Documentation**
   - Write comprehensive documentation explaining the design and implementation of the operator.
   - Include a guide on how to build, deploy, and test the operator in a Kubernetes cluster.
3. **Deployment Instructions**
   - Provide a Helm chart or Kubernetes manifests to deploy the operator and the CRDs.
   - Include instructions for deploying the operator to a local Minikube cluster or a remote Kubernetes cluster.
4. **Testing Instructions**
   - Include instructions on how to run the unit tests and end-to-end tests.
   - Provide example `ComputeNode` and `ComputeJob` resource manifests for testing different scenarios.

Evaluation Criteria

1. **Code Quality**: Clean, well-organized, and idiomatic Go code.
2. **Kubernetes Knowledge**: Correct use of Kubernetes API, resources, and best practices.
3. **Functionality**: The operator should fully manage the lifecycle of `ComputeNode` and `ComputeJob` resources as described.
4. **Testing**: Comprehensive unit and end-to-end tests covering key scenarios.
5. **Documentation**: Clear and detailed documentation and deployment instructions.

Submission

Please submit your solution as a zip file, including instructions on how to build, test, and run the code. Ensure that the code is not placed in any public repository.
Good luck!