🧩 Lab: Building a Custom Kubernetes Scheduler in Python

🎯 Learning Objectives

By the end of this lab, you will:

- Understand how the **scheduler** interacts with the **API Server**.
- Implement a **custom scheduler** that:
  - Finds **Pending Pods** with a given `schedulerName`.
  - Chooses a **Node** according to a scheduling policy.
  - **Binds** the Pod to that Node through the Kubernetes API.
- Compare **polling vs event-driven (watch)** models.
- Deploy your scheduler into a **kind cluster** and observe its behavior.

🧰 Environment Setup

Prerequisites

- Docker (latest)
- kind (`go install sigs.k8s.io/kind@latest`)
- kubectl
- Python 3.11+
- (Optional) VS Code or PyCharm for editing

Create and Inspect Cluster

```
kind create cluster --name sched-lab
kubectl cluster-info
kubectl get nodes
```

⚙️ Step 1 — Observe the Default Scheduler

1. Identify the running scheduler:

   ```
   kubectl -n kube-system get pods -l component=kube-scheduler
   kubectl -n kube-system logs -l component=kube-scheduler
   ```

2. Schedule a simple pod:

```
kubectl run test --image=nginx --restart=Never
kubectl get pods -o wide
```

> ✅ **Checkpoint 1:**
> Describe the path:
> `kubectl run` → Pod created → Scheduler assigns Node → kubelet starts Pod.

🧱 Step 2 — Project Setup

Initialize Project

```
mkdir py-scheduler && cd py-scheduler
python -m venv .venv && source .venv/bin/activate
pip install kubernetes==29.0.0
touch scheduler.py
```

Directory Structure

```
py-scheduler/
├── scheduler.py
├── Dockerfile
├── rbac-deploy.yaml
├── test-pod.yaml
└── requirements.txt
```

🧠 Step 3 — Implement the Polling Scheduler

Create a simple scheduler in **scheduler.py**:

```
from kubernetes import client, config
import time, math

def load_client():
    config.load_incluster_config()
    return client.CoreV1Api()

def bind_pod(api, pod, node):
    target = client.V1ObjectReference(kind="Node", name=node)
    meta = client.V1ObjectMeta(name=pod.metadata.name)
    body = client.V1Binding(target=target, metadata=meta)
    api.create_namespaced_binding(pod.metadata.namespace, body)

def choose_node(api):
```

```python
    nodes = api.list_node().items
    pods  = api.list_pod_for_all_namespaces().items
    min_cnt, pick = math.inf, nodes[0].metadata.name
    for n in nodes:
        cnt = sum(1 for p in pods if p.spec.node_name == n.metadata.name)
        if cnt < min_cnt:
            min_cnt, pick = cnt, n.metadata.name
    return pick

def main():
    api = load_client()
    while True:
        pods = 
api.list_pod_for_all_namespaces(field_selector="spec.nodeName=").items
        for pod in pods:
            if pod.spec.scheduler_name != "my-scheduler":
                continue
            node = choose_node(api)
            bind_pod(api, pod, node)
            print(f"Bound {pod.metadata.name} -> {node}")
        time.sleep(2)

if __name__ == "__main__":
    main()
```

> ✅ **Checkpoint 2:**
> Understand the control loop:
>
> - **Observe:** list unscheduled Pods
> - **Decide:** pick a Node
> - **Act:** bind the Pod

🐳 Step 4 — Build and Deploy

Dockerfile

```dockerfile
FROM python:3.11-slim
WORKDIR /app
COPY . .
RUN pip install kubernetes==29.0.0
ENTRYPOINT ["python", "scheduler.py"]
```

Build & Load Image

```bash
docker build -t my-py-scheduler:latest .
kind load docker-image my-py-scheduler:latest --name sched-lab
```

🔐 Step 5 — RBAC & Deployment

Create a file named **rbac-deploy.yaml**:

```yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: my-scheduler
  namespace: kube-system
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: my-scheduler-binding
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: system:kube-scheduler
subjects:
- kind: ServiceAccount
  name: my-scheduler
  namespace: kube-system
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-scheduler
  namespace: kube-system
spec:
  replicas: 1
  selector:
    matchLabels: {app: my-scheduler}
  template:
    metadata:
      labels: {app: my-scheduler}
    spec:
      serviceAccountName: my-scheduler
      containers:
      - name: scheduler
        image: my-py-scheduler:latest
```

Apply it:

```
kubectl apply -f rbac-deploy.yaml
kubectl -n kube-system get pods -l app=my-scheduler
```

🧪 Step 6 — Test Your Scheduler

Create a pod definition **test-pod.yaml**:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: test-pod
spec:
  schedulerName: my-scheduler
  containers:
  - name: pause
    image: registry.k8s.io/pause:3.9
```

Deploy and observe:

```
kubectl apply -f test-pod.yaml
kubectl get pods -o wide
kubectl -n kube-system logs deploy/my-scheduler
```

> ✅ **Checkpoint 3:**
> Your scheduler should log a message like:
> `Bound default/test-pod -> kind-control-plane`

🧩 Step 7 — Event-Driven Scheduler (Watch API)

Replace the polling loop with a **watch-based** approach:

```python
import argparse, math
from kubernetes import client, config, watch

# TODO: load_client(kubeconfig) -> CoreV1Api
#  - Use config.load_incluster_config() by default, else
config.load_kube_config()

# TODO: bind_pod(api, pod, node_name)
#  - Create a V1Binding with metadata.name=pod.name and
target.kind=Node,target.name=node_name
#  - Call api.create_namespaced_binding(namespace, body)

# TODO: choose_node(api, pod) -> str
#  - List nodes and pick one based on a simple policy (fewest running
pods)

def main():
    parser = argparse.ArgumentParser()
    parser.add_argument("--scheduler-name", default="my-scheduler")
    parser.add_argument("--kubeconfig", default=None)
    args = parser.parse_args()
```

```python
    # TODO: api = load_client(args.kubeconfig)

    print(f"[watch-student] scheduler starting… name=
{args.scheduler_name}")
    w = watch.Watch()
    # Stream Pod events across all namespaces
    for evt in w.stream(client.CoreV1Api().list_pod_for_all_namespaces,
_request_timeout=60):
        obj = evt['object']
        if obj is None or not hasattr(obj, 'spec'):
            continue
        # TODO: Only act on Pending pods that target our schedulerName
        #  - if obj.spec.node_name is not set and obj.spec.scheduler_name
== args.scheduler_name:
        #       node = choose_node(api, obj)
        #       bind_pod(api, obj, node)
        #       print(...)

if __name__ == "__main__":
    main()
```

> ✅ **Checkpoint 4:**
> Compare responsiveness and efficiency between **polling** and **watch** approaches.

🧩 Step 8 — Policy Extensions

1. **Label-based node filtering**

   ```python
   nodes = [n for n in api.list_node().items
            if "env" in (n.metadata.labels or {}) and
   n.metadata.labels["env"] == "prod"]
   ```

2. **Taints and tolerations** Use `node.spec.taints` and `pod.spec.tolerations` to filter nodes before scoring.

3. **Backoff / Retry** Use exponential backoff when binding fails due to transient API errors.

4. **Spread policy** Distribute similar Pods evenly across Nodes.

> ✅ **Checkpoint 5:**
> Demonstrate your extended policy via pod logs and placement.

🔍 Observability and Debugging

Command	Description

Command	Description
`kubectl get pods -o wide`	Check node assignments
`kubectl get events --sort-by=.lastTimestamp`	Watch scheduling events
`kubectl describe pod test-pod`	Inspect scheduling status
`kubectl -n kube-system logs deploy/my-scheduler -f`	View scheduler logs

🧹 Cleanup

```
kubectl delete -f test-pod.yaml
kubectl delete -f rbac-deploy.yaml
kind delete cluster --name sched-lab
```

🧾 Deliverables

Each group (3 students) must submit:

1. **Project files** (`scheduler.py`, `Dockerfile`, `rbac-deploy.yaml`, `test-pod.yaml`). A tgz git repo is preferred.
2. **Logs/screenshots** showing successful scheduling.
3. A short report (~1 page) describing:
    - Node selection logic.
    - Observations of scheduling behavior.
    - Differences between polling and watch models.

🧮 Evaluation Criteria

Criterion	Weight	Description
Working scheduler	30%	Scheduler deploys and binds pods correctly
Correct binding logic	25%	Only handles pods with correct schedulerName
Policy extension	25%	Custom filtering/scoring logic implemented
Reflection/report quality	20%	Explains results and design choices clearly

---

🧠 Reflection Discussion

- Why is it important that your scheduler writes a **Binding** object instead of patching a Pod directly?
- What are the trade-offs between polling vs event-driven models?
- How do **taints and tolerations** interact with your scheduling logic?
- What are real-world policies you could implement using this framework?

🔬 Bonus Exercise (Optional)

Implement a **taint-aware scheduler extension** using this helper:

```python
def node_tolerates_taints(node, pod):
    taints = node.spec.taints or []
    tolerations = pod.spec.tolerations or []
    if not taints:
        return True
    for taint in taints:
        tolerated = any(
            tol.key == taint.key and
            (tol.effect == taint.effect or tol.effect is None) and
            (tol.operator == "Exists" or tol.value == taint.value)
            for tol in tolerations
        )
        if not tolerated:
            return False
    return True
```

> 🧭 **End of Lab**