Kubernetes Internals: Architecture and Scheduling Deep Dive

Motivation

- Debug complex production issues
- Design operators and custom controllers
- Extend scheduling for research/policy
- Tune performance, scalability, HA

A Distributed Control System

- Declarative desired state
- Control loops drive convergence
- Consensus (etcd, Raft) and leader election
- Event-driven/Reactive updates via Watches

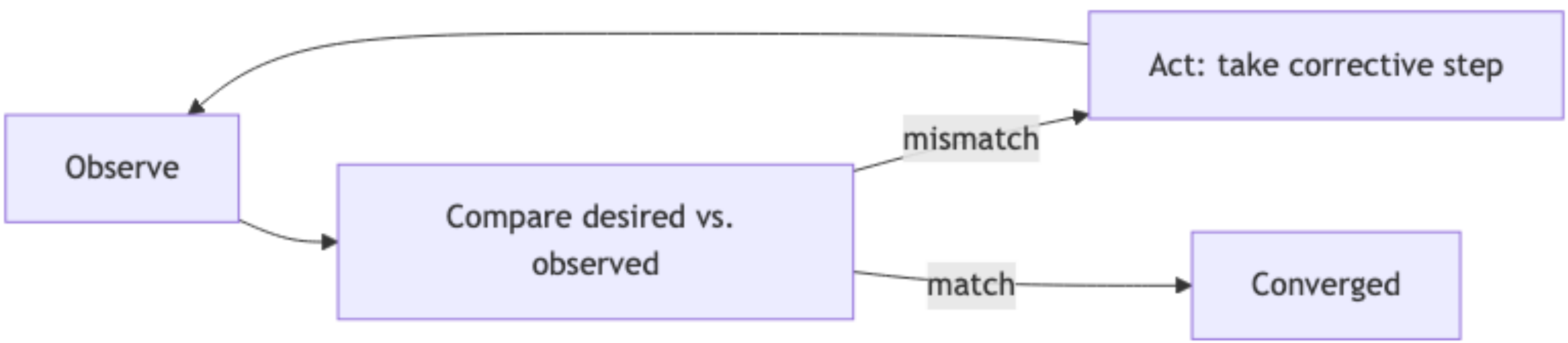
Orchestrator… of what

- Atomic workload: the POD
  - Specification of something to do
- Atomic unit of placement: the Node
- Main goal: place PODS in Nodes

Declarative Management Model

- Users submit specs to the API server
- Controllers and scheduler reconcile toward spec
- Source of truth: persisted state in etcd

Control Loops: Observe → Compare → Act



Key Principles from D.S.

- Fault tolerance via replication & leader election
- Scalability by horizontal partitioning
- Availability through retries/backoff
- Consistency model tuned via Raft (etcd)

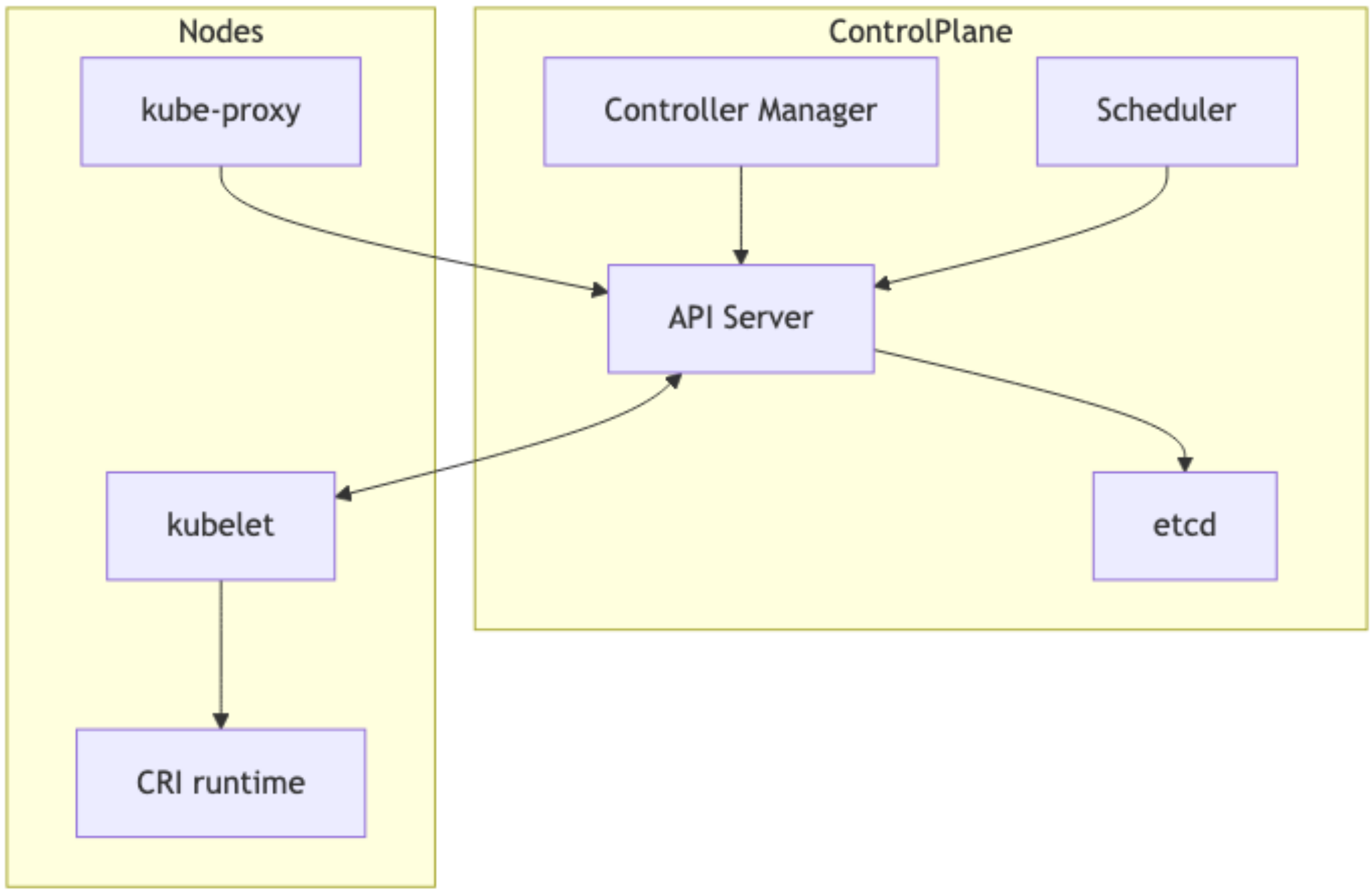
Agenda

- Architecture overview (control vs data plane)
- API server and etcd internals
- Controller pattern
- Scheduler framework and plugins
- Lab: build a custom scheduler in Go

Interaction Overview

- API server fronts all interactions
- etcd stores cluster state (strong consistency)
- Controllers reconcile; scheduler places pods
- kubelet, runtime, and CNI enact decisions

High-level Component Diagram



Key Takeaways (Intro)

- Kubernetes = cooperating control loops
- API Server is choke point and source of truth
- etcd provides strong consistency; watches drive reactivity

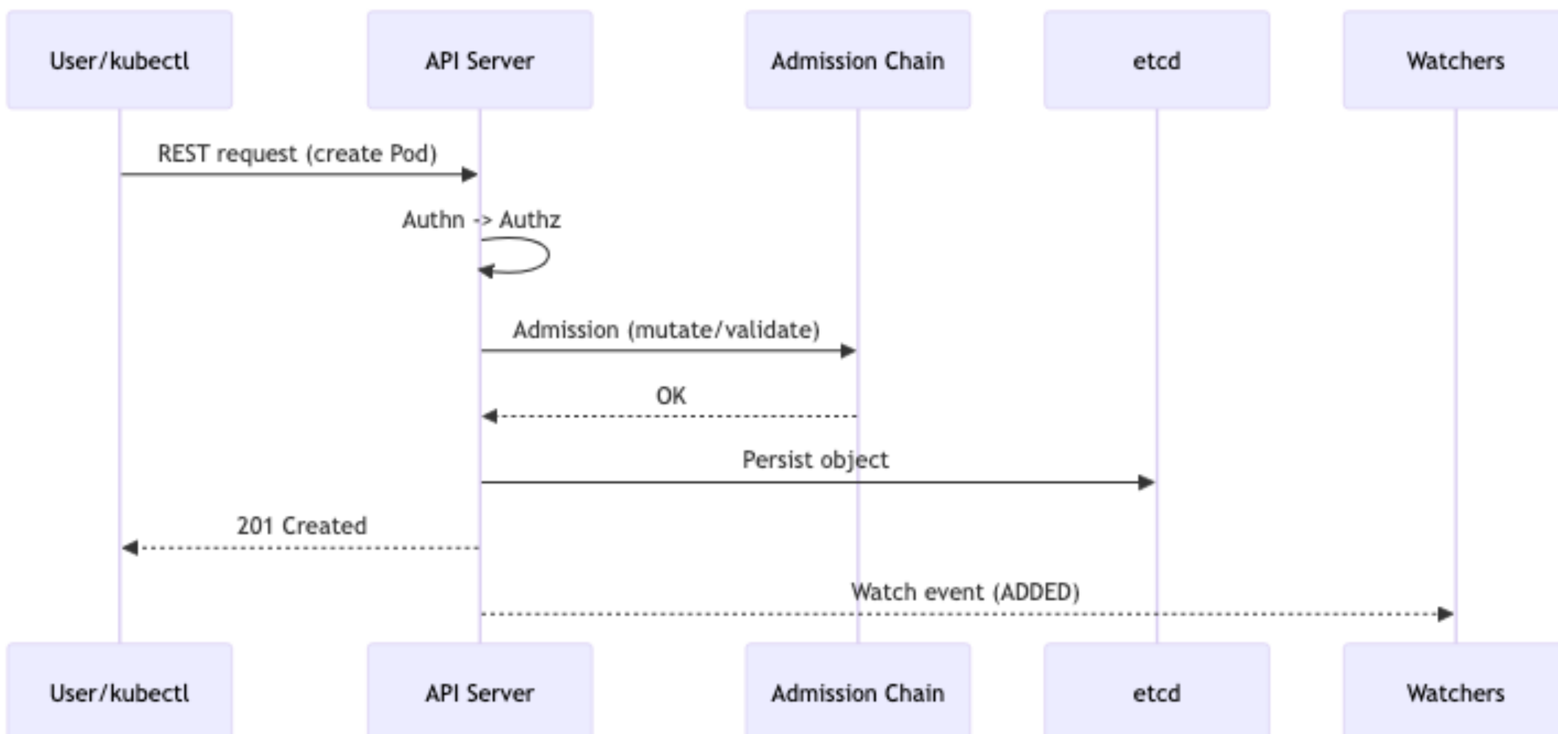
Control Plane Components

- API Server: validation, authn/z, admission
- etcd: strongly consistent KV store
- Controller Manager: reconcilers
- Scheduler: placement decisions
- Cloud Controller Manager: cloud integration

API Server Responsibilities

- Authentication/Authorization/Admission
- Versioned API groups and resources
- Serialization (JSON/Protobuf) and storage
- Watches for clients (controllers, scheduler)

API Request Flow



Admission Controllers (Examples)

- NamespaceLifecycle, LimitRanger, ResourceQuota
- Mutating/Validating Webhooks for custom policies

Storage and Keys in etcd

- Keys: /registry/<resource>/<ns>/<name>
- Protobuf for performance
- Compaction and snapshots

Watches & Event Delivery

- Long-lived watch streams reduce polling
- ResourceVersion for resumable streams
- Backpressure and relist strategies

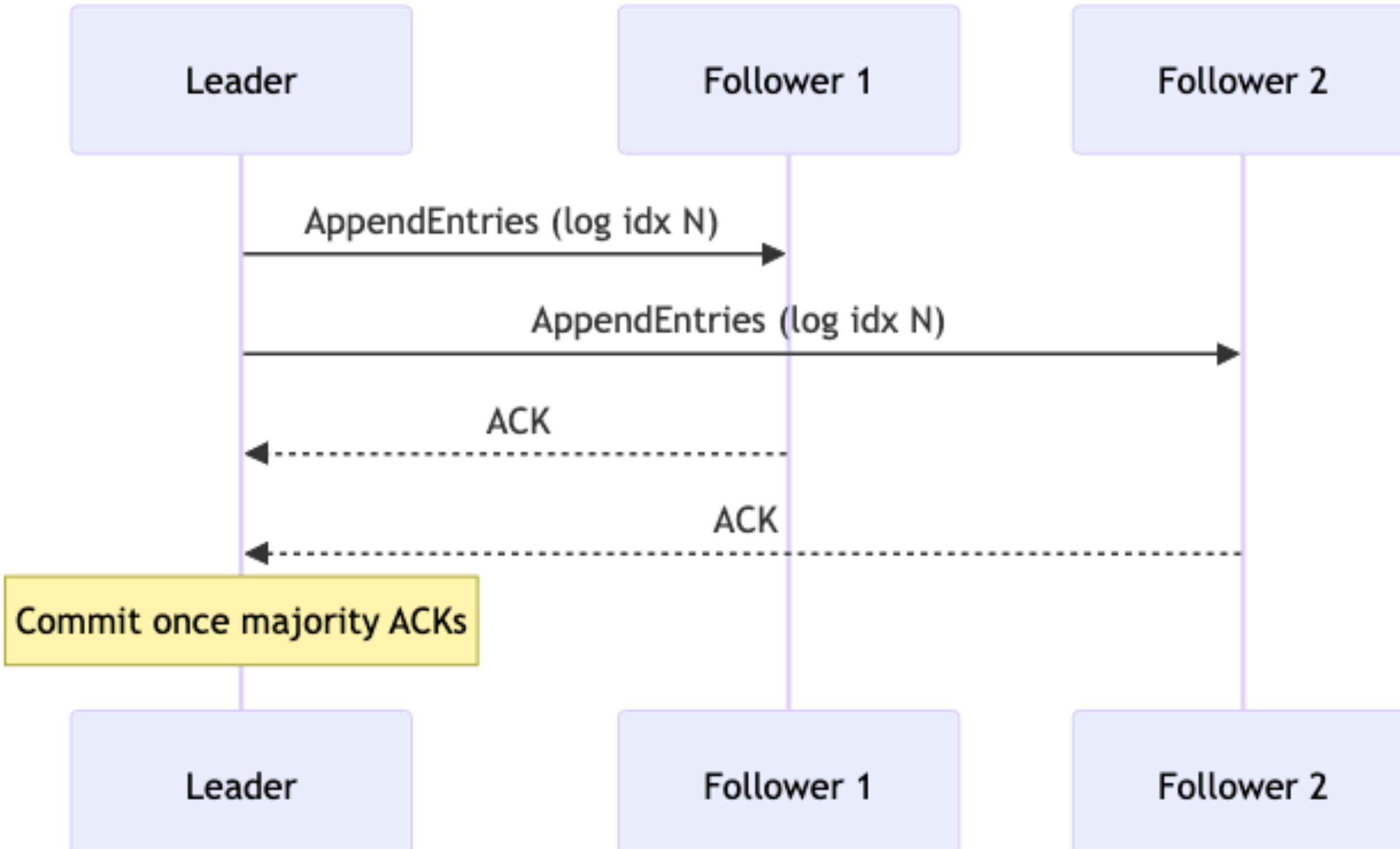
etcd Deep Dive

- Linearizable reads/writes (strong consistency)
- Raft consensus: log replication; single leader
- Quorum writes; follower read via leader lease

etcd Cluster Sizing

- 3 or 5 members for HA
- Odd numbers maximize quorum safety
- Latency-sensitive: network & storage

Raft at a Glance



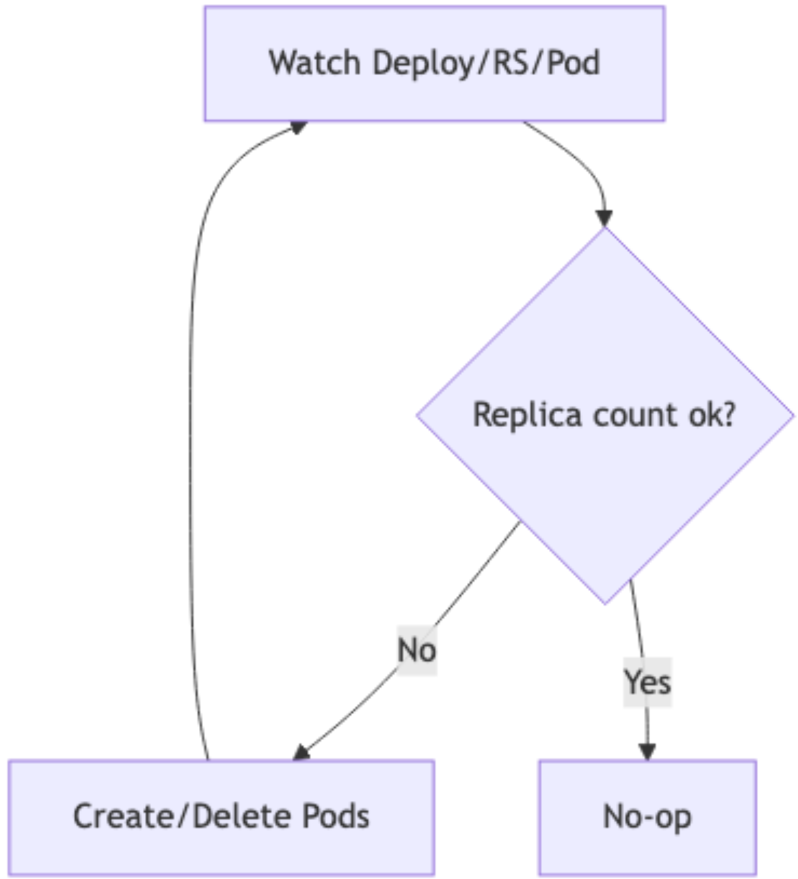
Control Loops: Controller Manager

- ReplicaSet, Job, Node, EndpointSlice, etc.
- Each controller owns a resource behavior
- Communicate only through the API server

Reconciliation Loop Pattern

- Observe current -> compare with desired -> act
- Idempotent actions; safe retries
- Workqueues, rate-limits, backoff

ReplicaSet Controller Flow



Cloud Controller Manager

- Abstracts interacting with the infrastructure provider
- Node discovery & lifecycle
- LoadBalancer provisioning
- Persistent volume integration
- Separate process/pod in the control plane
- Plugin-driven (per IaaS provider)

CCM example flow

- The Kubernetes API server creates the Service object.
- The CCM notices this event
- The Service Controller within the CCM calls the AWS API to create an Elastic Load Balancer (ELB)
- The ELB gets associated with the worker nodes running the service Pods
- The Service’s status.loadBalancer.ingress field is updated with the ELB’s DNS name or IP

Control Plane: HA Strategies

- Multiple API servers behind LB
- Leases for leader election
- Stateless binaries; rehydrate from API

Scaling the Control Plane

- Have several réplicas of API Server, and place them behind a Load balancer
- API server: tune etcd, cache, admission
- Shard heavy webhooks; reduce sync storms
- etcd compaction/defrag schedules
- Place bulky configuration data elsewhere

Observability: Control Plane

- Component logs (structured)
- Prometheus metrics (apiserver_, scheduler_)
- Health probes: /livez, /readyz

Security: Control Plane

- TLS everywhere, mTLS between components
  - Network policies diminish use for internal TLS
- RBAC: roles and bindings; least privilege
- Audit logs for forensics

Control Plane Recap

- API server as hub; etcd as truth
- Controllers are reconcilers
- Watches enable event-driven behavior
- Periodic polls, avoid missing updates

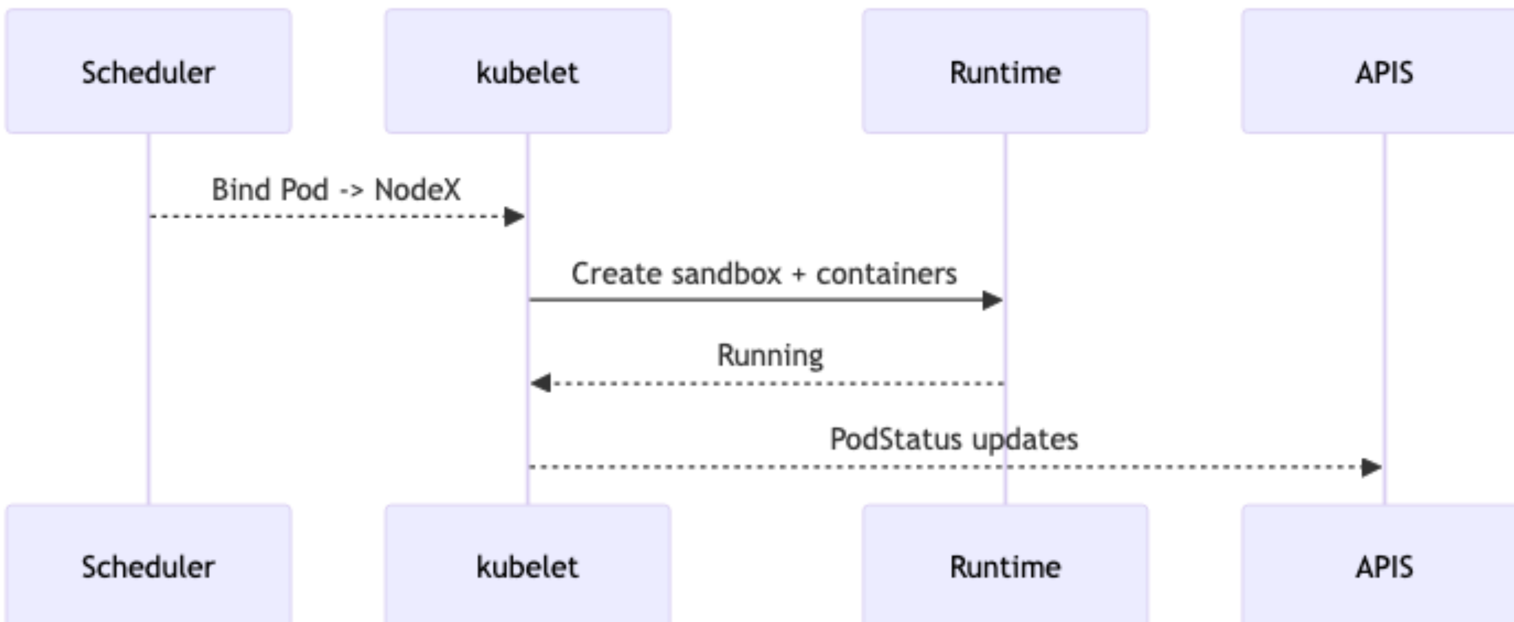
Data Plane Components

- kubelet: node agent and Pod lifecycle
- Container runtime: CRI (containerd, CRI-O)
- kube-proxy: services via iptables/IPVS

kubelet Responsibilities

- Node registration & heartbeats
- Pod lifecycle (create/start/monitor/restart)
- CNI/CSI orchestration via CRI/CSI

Pod Lifecycle (Node Perspective)



kube-proxy and Services

- Maintains node-level network rules for Service load balancing
- Redirect traffic from Service IPs/ports to backend Pod IPs
- ClusterIP, NodePort, LoadBalancer types
- iptables or IPVS for service routing
- EndpointSlice for scale

Example flow

- Pod A sends traffic to 10.96.0.10:80 (the Service ClusterIP)
- The Linux kernel consults the iptables/IPVS rules installed by kube-proxy.
- The packet is NAT’d to one of the Pod IPs (say 10.244.2.7:8080).
- The packet is routed through the CNI network to the node running that Pod.
- The Pod receives the packet as if it came directly from Pod A.

Container Runtime Interface (CRI)

- gRPC interface kubelet ↔ runtime
- Image management, sandbox, containers
- RuntimeClass for alt runtimes (gVisor, Kata)

Networking Deep Dive (CNI)

- Pod network vs service network
- Common CNIs: Calico, Flannel, Cilium
- Encapsulation vs routing vs eBPF approaches

Node Health and Eviction

- Node controller marks NotReady
- Pod eviction and rescheduling
- Taints/tolerations for policies

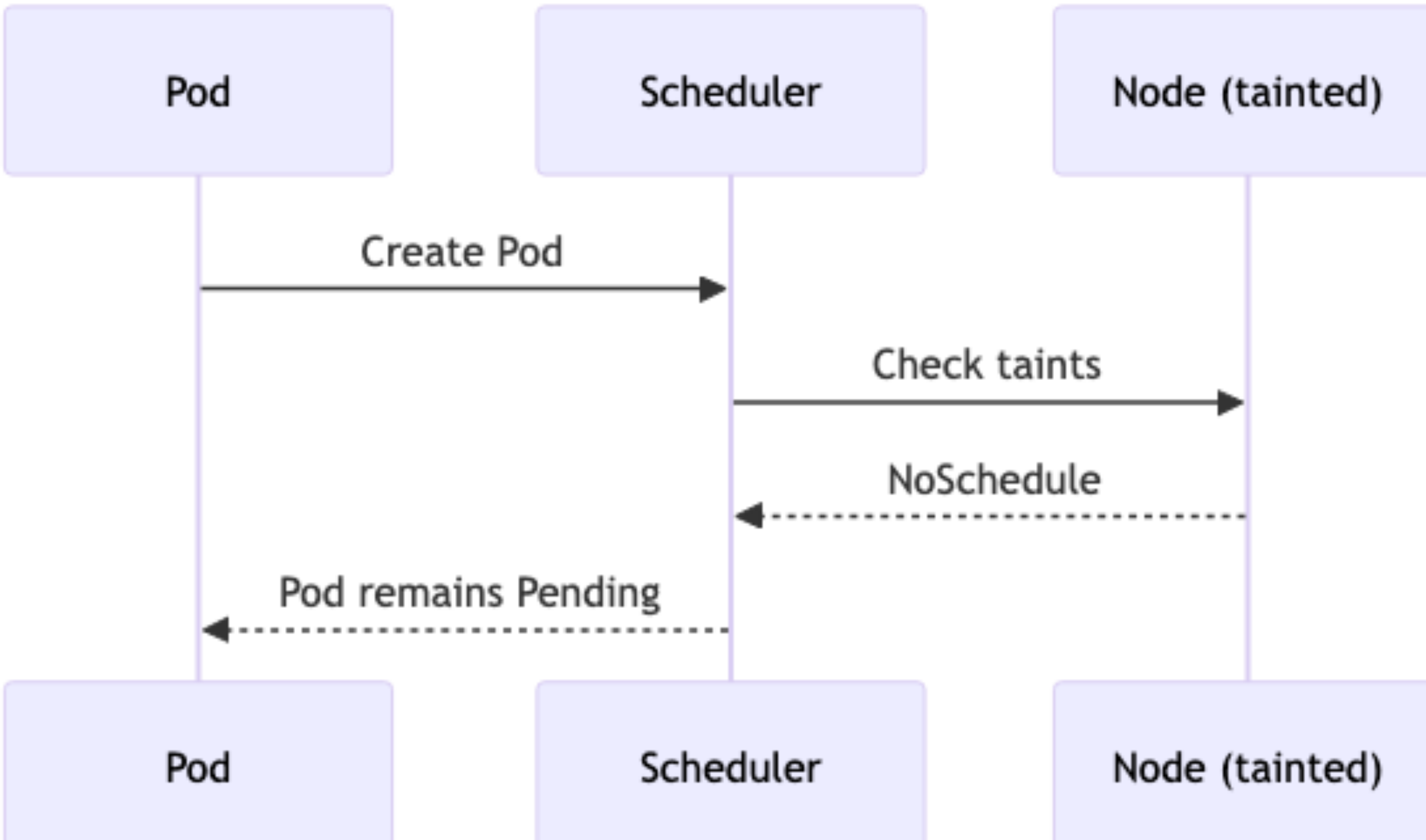
Taints and Tolerations: Why They Matter

- Control pod placement on specific nodes using declarative policies.
- Use cases:
- - GPU or ML training nodes dedicated to heavy workloads.
- - Nodes reserved for system daemons or maintenance.
- Pods must tolerate taints to be scheduled onto tainted nodes.

Understanding Taints

- A taint is applied to a Node to discourage or prevent regular Pods.
- Syntax: kubectl taint nodes <node> <key>=<value>:<effect>
- Effects:
- - NoSchedule: new Pods are prevented.
- - PreferNoSchedule: scheduler avoids but not enforced.
- - NoExecute: also evicts existing Pods without toleration.

Taint Evaluation Sequence



Understanding Tolerations

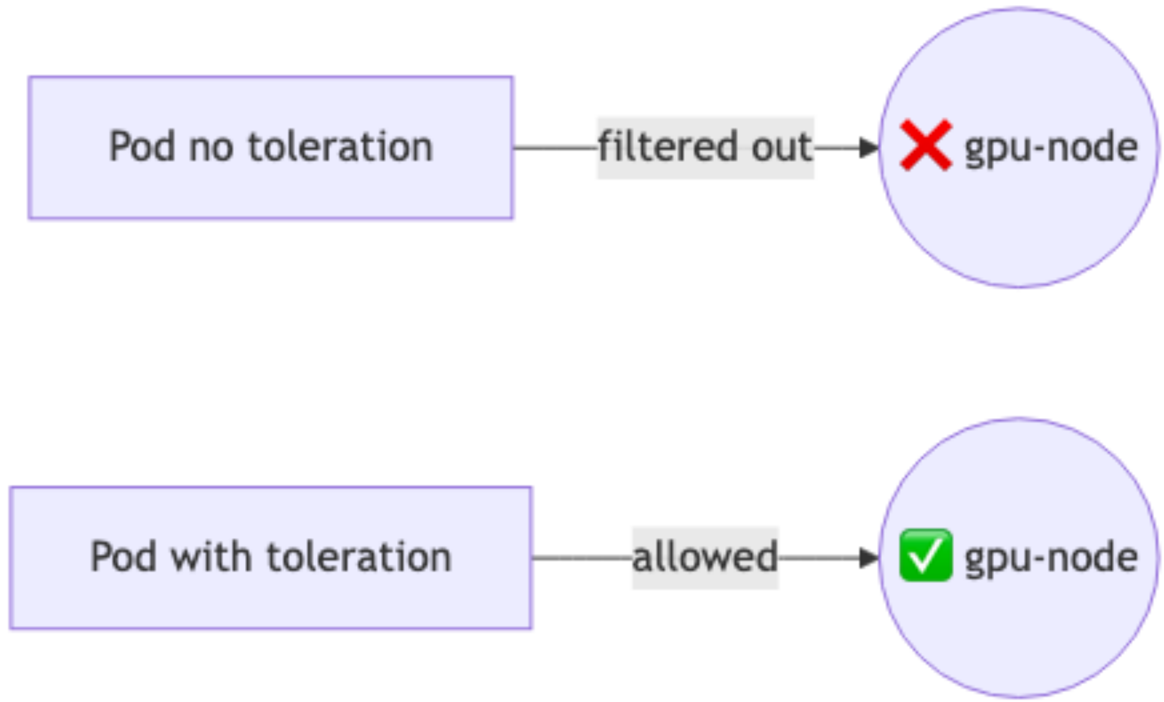
- A toleration on a Pod allows it to ignore specific node taints.
- Example:
- tolerations:
-   - key: dedicated
-     operator: Equal
-     value: gpu
-     effect: NoSchedule
- Pods with this toleration can run on nodes tainted with dedicated=gpu:NoSchedule.

Example: Matching Taint and Toleration

```
# Mark a node as GPU-only
kubectl taint nodes gpu-node dedicated=gpu:NoSchedule

# Pod that tolerates the taint
apiVersion: v1
kind: Pod
metadata:
    name: gpu-job
spec:
    schedulerName: my-scheduler
    tolerations:
    - key: "dedicated"
        operator: "Equal"
        value: "gpu"
        effect: "NoSchedule"
    containers:
    - name: compute
        image: nvidia/cuda:12.0-base
```

Scheduling Outcomes



Adding Taint/Toleration Filtering to Your Scheduler

```
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 = False
        for tol in tolerations:
            if tol.key == taint.key and (tol.effect == taint.effect or tol.effect is None):
                if tol.operator == "Exists" or tol.value == taint.value:
                    tolerated = True
                    break
        if not tolerated:
            return False
    return True

def choose_node(api, pod):
    nodes = api.list_node().items
    valid = [n for n in nodes if node_tolerates_taints(n, pod)]
    if not valid:
        raise RuntimeError("No schedulable nodes (taints not tolerated)")
    return valid[0].metadata.name
```

Data Plane Faults

- Container crash loops -> kubelet restarts
- Network partitions -> readiness fails
- Disk pressure -> eviction manager acts

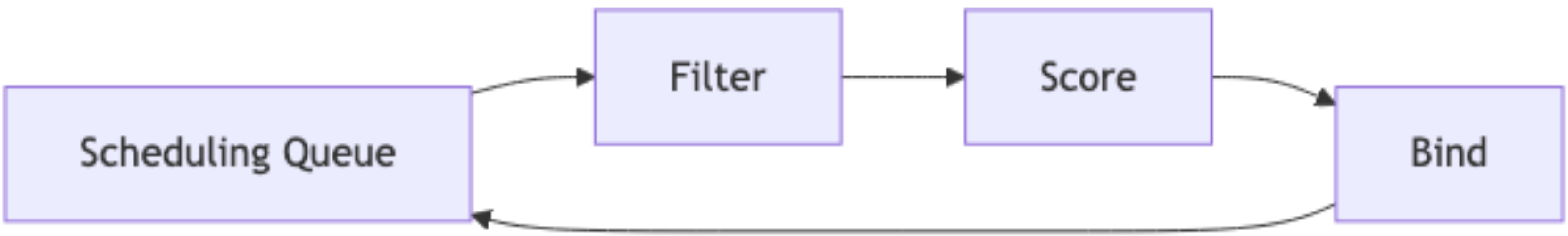
Data Plane Recap

- Execution side of the cluster
- kubelet translates specs to actions
- Network and storage plugins provide plumbing

Scheduler Overview

- Async controller that binds Pods to Nodes
- Goals: constraints fit, optimization, fairness
- Framework offers hook points for policy

Scheduling Cycle



Filtering Phase (Feasibility)

- Node readiness & taints/tolerations
- Resource requests vs capacity
- Node Affinity/Pod Affinity–AntiAffinity
- Topology & constraints

Scoring Phase (Ranking)

- LeastRequestedPriority & BalancedAllocation
- Spread across topology keys
- Custom scoring plugins for domain-specific goals

Binding & Permit/Reserve

- Reserve locks node resources during cycle
- Permit gates binding (pre-binding checks)
- Bind writes Pod->Node binding to API

Scheduler Framework Architecture

- Plugins invoked at extension points
- Per-profile configuration; multi-profile clusters
- In-tree and out-of-tree plugins

Scheduler Cache & State

- Node/Pod snapshots for quick evaluation
- Invalidation via watch events
- Optimizations: precomputation & parallelism

Scheduler Extenders

- HTTP-based out-of-process filters/scores
- Use-cases: GPU placement, external constraints
- Trade-offs: latency, reliability

Multi-Scheduler Clusters

- Pods choose via spec.schedulerName
- Isolation of policies by namespace/app
- Default scheduler ignores non-default pods

Pod Targeting a Custom Scheduler (YAML)

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

Performance & Debugging

- Increase verbosity: --v=4 (or higher)
- Observe events to watch preemption/backoff
- Prometheus metrics: scheduler_*, framework_*

Failure & HA

- Leader election ensures active scheduler
- Pending pods wait; no data loss
- Stateless; restart safe

Advanced Concepts

- Preemption for priority classes
- Scheduling profiles per workload class
- Scoring normalization and weights

Scheduler Deep Dive Recap

- Filter -> Score -> (Preempt?) -> Bind
- Framework = composable policy engine
- Extenders and multi-scheduler for specialization

Lab: Building a Custom Kubernetes Scheduler in Python

- Goal: Implement a minimal scheduler using the Kubernetes Python client.
- Observe how Pods move from Pending → Bound state.
- Understand and extend scheduling logic (polling & watch).
- Run inside a kind cluster as a Deployment.

Environment Setup Commands

```
# Prerequisites:
# Docker, kind, kubectl, Python 3.11
kind create cluster --name sched-lab
kubectl cluster-info
kubectl get nodes
```

Step 1: Project Setup

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

Step 2: Minimal Polling Scheduler

```python
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):
    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()
```

Step 3: Package and Deploy

```
# Dockerfile
FROM python:3.11-slim
#...

# Build and load into kind
docker build -t my-py-scheduler:latest .
kind load docker-image my-py-scheduler:latest --name
sched-lab
```

Step 4: RBAC + Deployment 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
```

Step 5: Test with a Custom Pod

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

Watch skeleton

```
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()

    # 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()
```

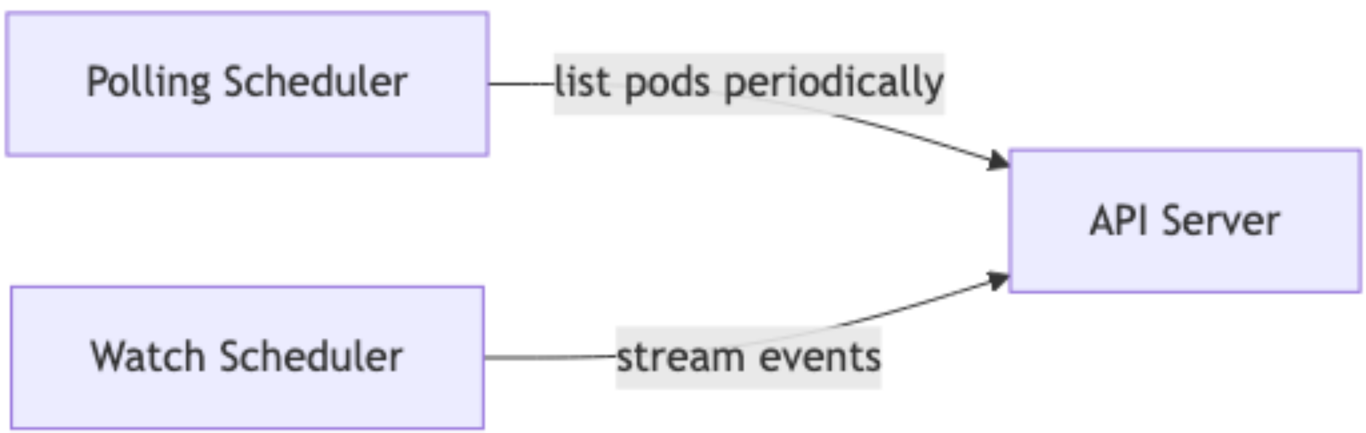
Validation Steps

- kubectl apply -f test-pod.yaml
- kubectl get pods -o wide
- kubectl -n kube-system logs deploy/my-scheduler -f
- Observe log output: Bound <namespace>/<pod> -> <node>

Step 6: Event-Driven Version (Watch API)

- Replace polling loop with: watch.Watch().stream(api.list_pod_for_all_namespaces)
- Respond only to ADDED or MODIFIED events.
- Filter for Pods where spec.nodeName is empty and schedulerName matches.
- Bind immediately after node selection.

Polling vs Watch-Based Scheduling



Step 7: Extensions & Experiments

- Implement label-based filtering: select nodes with specific labels (e.g., env=prod).
- Integrate taints/tolerations filtering.
- Add naive backoff for failed bindings.
- Compare performance of polling vs watch-based schedulers.

Step 8: Reflection & Deliverables

- Each group submits:
- - Working Python watch scheduler project (code + YAMLs).
- - Screenshots/logs of successful scheduling.
- - A short report explaining their selection logic and observations.

HA Control Plane Review

- Multi-API server with LB
- Leases for scheduler/controller HA
- etcd quorum: 3–5 members

Beyond the Scheduler

- Operators for domain logic (CRDs + control loops)
- Admission webhooks for policy
- Custom resources and controllers

Research Directions

- AI-assisted and predictive scheduling
- Energy-aware & carbon-intensity aware placement
- Heterogeneous clusters: GPUs, FPGAs, edge

Key Takeaways

- Kubernetes is a set of cooperating control loops
- API server centralizes state and access
- Scheduler framework enables policy innovation

References & Next Steps

- Official docs and SIG-Scheduling design docs
- etcd / Raft resources
- Proceed to extended lab or plugin-based scheduler