Pod Notes

1) Quick summary (one-liner)

A **Pod** is the smallest deployable unit in Kubernetes: one or more co-scheduled containers that share network namespace, IPC, and volumes. Creation flows from the user/client (kubectl) to the API server, may be mutated/validated, is persisted to etcd, scheduled by the scheduler (assigns a node), and then the kubelet on the chosen node observes the Pod object and drives the container runtime to create containers.

---

2) Control plane & data plane — high level roles (short)

- **API Server (`kube-apiserver`)** — single entry point for all REST operations; validates, authenticates, authorizes, runs admission controllers, stores objects in etcd, serves watch streams.
- **etcd** — consistent key-value store; persistent source of truth for cluster state (e.g., `/registry/...` keys).
- **Controller Manager (`kube-controller-manager`)** — runs controllers (ReplicationController/ReplicaSet, Deployment, Job, Node controller, Endpoint controller, ServiceAccount controller, etc.).
- **Scheduler (`kube-scheduler`)** — assigns unscheduled Pods to Nodes (by writing Pod.spec.nodeName or creating a Binding).
- **Cloud Controller Manager** — cloud-provider specific controllers (routes, load balancers, node lifecycle in cloud).
- **kubelet (on each node)** — node agent: watches API server for Pod objects assigned to its node, starts/stops containers via CRI, reports status.
- **Container Runtime (CRIo/containerd/dockerd) + OCI runtime (runc/crun)** — actually pulls images and creates containers.
- **kube-proxy** — programs networking (iptables/ipvs) for Services on each node.
- **CNI plugins** — set up Pod network interfaces and IPs.
- **CSI drivers** — attach/mount volumes on nodes when Pods request persistent volumes.

---

3) Pod object basics (what is stored in etcd)

Pod is a group of one or more containers
All containers in the Pod are
  – co-located
  – co-scheduled

A Pod object is a Kubernetes API resource (`api/v1`), typically stored under a path like:

```
/registry/pods/<namespace>/<podName>
```

Key important fields:

- `metadata.uid`, `metadata.resourceVersion`, `metadata.creationTimestamp`
- `spec` — containers, initContainers, volumes, nodeName (initially empty when unscheduled), tolerations, affinity, etc.
- `status` — phase, conditions, containerStatuses (updated by kubelet)
  `resourceVersion` is used for optimistic concurrency + watch semantics.

---

4) Step-by-step creation flow (detailed sequence)

I'll show a numbered sequence and then expand each step.

Sequence (short)

1. `kubectl apply -f pod.yaml` → `kubectl` calls API server (REST).
2. API server authenticates/authorizes request.
3. Admission controllers run (mutating → validating). Mutations may change spec (e.g., inject sidecar).
4. API server persists Pod object to etcd (write).
5. API server returns created Pod object to client.
6. `kube-scheduler` sees unscheduled Pod (via watch or list).
7. Scheduler filters & scores nodes; chooses node → writes binding (sets `spec.nodeName` or creates a `Binding`).
8. API server persists that update to etcd. A watch event is emitted.
9. Kubelet on the chosen node sees Pod assigned to it (via watch/list) and enqueues it to the pod sync loop.
10. Kubelet creates the Pod: calls CRI to create sandbox (pause or infra container), then pulls images and creates containers.
11. Kubelet reports PodStatus back to API server (status subresource patch/write). etcd updated.
12. kube-proxy / CNI / CSI finalize network and volume attachments as needed; Services/endpoints updated; Pod becomes Ready when readyProbe passes.
13. Controllers (ReplicaSet/Deployment) continue to observe and reconcile desired state.

Now expanded details:

Step 1 — Client submits Pod to API Server

Command example:

```
kubectl apply -f mypod.yaml
# or
kubectl create -f mypod.yaml
```

`kubectl` performs an HTTP REST call to `POST /api/v1/namespaces/<ns>/pods` (or PATCH for apply). The request includes the Pod manifest in JSON/YAML.

Step 2 — API Server: authN / authZ

- **Authentication**: API server checks the client identity (client cert / OIDC token / service account token).
- **Authorization**: RBAC or ABAC decides whether the caller can create the Pod.

If auth fails, it returns 401/403 and the flow stops.

Step 3 — Admission controllers (mutating → validating)

- **Mutating Admission Webhooks & Built-in Mutating Controllers** run first. They can:
  - Inject defaults.
  - Inject sidecars (e.g., Istio, Linkerd) via MutatingAdmissionWebhook.
  - Add `imagePullSecrets`, default resource limits, security policies, etc.
- **Validating Admission Webhooks & built-ins** run after mutations and can reject.
- Common built-ins: `NamespaceLifecycle`, `LimitRanger`, `MutatingAdmissionWebhook`, `ValidatingAdmissionWebhook`, `ResourceQuota`, `PodSecurity` (or PodSecurityPolicies in older clusters).

Important: the object that gets persisted may be different from the user's original YAML because of mutations (e.g., a sidecar container added).

Step 4 — API Server writes Pod to etcd

- The API server marshals the final Pod object into JSON and writes it to etcd under `/registry/pods/<namespace>/<name>`.
- etcd assigns a new `modRevision`. The Pod gets a `metadata.resourceVersion` that reflects storage version.
- The etcd write is strongly consistent (linearizable) — once the write returns success, other readers that request a current read will see it.

Note: API server also keeps an in-memory cache and responds to other components (controllers/scheduler) via watches.

Step 5 — The API Server responds to the client

- `kubectl` receives success and prints summary. But the Pod is *unscheduled* if `spec.nodeName` not set.

Step 6 — Scheduler notices unscheduled Pod

- The scheduler watches the API for Pods where `spec.nodeName` is empty and `status.phase` is not failed/succeeded.

- The scheduler uses **two phases**:
    1. **Filtering** (predicates): remove nodes that cannot run the pod (resource requests, taints & tolerations, nodeSelectors, affinity, node disk/capacity, port conflicts, volume constraints, topology).
    2. **Scoring**: rate the remaining nodes (binpacking, spread, custom scheduler extenders, plugin scores).
- Scheduler may also consider Pod priority and preemption: if no node fit, it may preempt lower priority pods on nodes to free resources.

Scheduling decision results in a **Bind** operation.

Step 7 — Scheduler writes the binding / sets `spec.nodeName`

Two ways historically:

- **Create a `Binding` subresource**: `POST /api/v1/namespaces/<ns>/pods/<pod>/binding` with `{ "target": { "apiVersion": "v1", "kind": "Node", "name":"node-1" }}` — API server will set `spec.nodeName` for the Pod.
- **Patch** the Pod object to set `spec.nodeName` (less common in modern schedulers).

This write goes through API server → etcd and is subject to admission (some admission controllers may block binding writes).

The write increments `resourceVersion` and emits a watch event for the Pod.

Step 8 — Watchers receive the event (kubelet sees assignment)

- Kubelet(s) watch Pods (they typically watch all Pods and filter by `spec.nodeName == myNode` locally).
- When kubelet sees a Pod assigned to its node, it enqueues it to the pod worker queue.

Note: There may be a slight delay between scheduler binding and kubelet seeing it due to watch propagation, but API server keeps consistent state.

Step 9 — Kubelet's pod sync loop (create Pod)

Kubelet processes the Pod in its sync loop:

1. **Admission & Validation at kubelet**: Kubelet checks Pod's security (PSP/PSA), volumes, and local node features. It also uses the container runtime interface (CRI).
2. **Prepare volumes**: if volumes are required, kubelet coordinates with CSI/attach/detach controllers (for attachable volumes) or mounts local volumes.
3. **Setup networking**: kubelet calls the CNI plugin to allocate a Pod IP and set up veth pair / network namespace.
    - The CNI result (pod IP) is stored in the PodStatus `podIP`.
4. **Create Pod sandbox**: kubelet calls CRI `RuntimeService.RunPodSandbox` — this creates the infra container / pause container that holds the network namespace.

5. **Pull images**: kubelet (via CRI) pulls images `ImageService.PullImage`.
6. **Create containers**: kubelet calls `RuntimeService.CreateContainer` for each container, then `StartContainer`.
7. **PostStart hooks / init containers**: init containers run sequentially; `postStart` lifecycle hooks are invoked inside the container runtime.
8. **Set status**: kubelet writes PodStatus updates back to the API (via the `status` subresource — these are separate writes than the Pod `spec` write).
    - Status updates include `phase` (Pending/Running/Failed), `containerStatuses` (image, state waiting/running/terminated), `podIP`.

Kubelet keeps working to ensure Pod remains desired (restarts failed containers according to restartPolicy). If containers crash repeatedly, kubelet updates status and controllers reconcile.

Step 10 — Readiness & Liveness probes and Endpoints

- **Readiness probe**: when it passes, kubelet marks `containerStatuses[].ready = true`; the Endpoints controller (or EndpointSlice controller) re-calculates endpoints and adds the Pod to the Service endpoints. Until ready, the Pod doesn't receive Service traffic.
- **Liveness probe**: failing liveness probe causes kubelet to restart container per restart policy.
- These changes are reflected as Status updates in the Pod object in etcd.

Step 11 — Controllers & Services notice Pod state

- ReplicaSet/Deployment observes Pod creation via informers and updates `status.replicas` etc. If more/less Pods than desired, the controller creates or deletes Pods.
- Service controllers add Pod IP to Endpoints/EndpointSlices as Pod becomes ready.

---

5) What happens in etcd — deeper detail and examples

etcd stores the authoritative serialized object. Typical key layout (examples simplified):

`/registry/pods/<namespace>/<podName> -> Pod JSON (spec, status)`

Storage events:

- **Create**: API server `PUT` creates key with createRevision & modRevision. The object stored contains `spec` and initial `status` (usually empty except maybe conditions).
- **Update (binding)**: Scheduler writes nodeName → modRevision increments.
- **Status updates**: kubelet patches the `status` subresource (API server writes to etcd) — status changes update `resourceVersion` again.
- **Watches**: Components (scheduler, kubelet, controllers) maintain watches on etcd via API server; watch streams send events: ADDED, MODIFIED, DELETED with `resourceVersion` for each event.
- **Optimistic concurrency**: Writes include preconditions (e.g., resourceVersion match) so stale

writes are rejected (409). This prevents races (scheduler vs some other actor).

- **Compaction & snapshot**: etcd compacts old revisions; API server may store different storage versions but etcd compaction removes old history. Regular backups/snapshots recommended.
- **Leader election / leases** (kube-controller-manager uses Lease objects stored in etcd to coordinate leader among replicas).

Illustration (example sequence of keys/versions):

1. `POST pod` → etcd key created `/registry/pods/ns1/pod1` -> resourceVersion `10`.
2. `Scheduler binds` → update `spec.nodeName` -> resourceVersion `12`.
3. `Kubelet status update` -> `status.phase=Running`, `podIP=10.244.1.5` -> resourceVersion `14`.
   Each update is an etcd write; watchers receive MODIFIED events with those resourceVersion values.

---

6) Important subresources & concurrency rules

- **status subresource**: kubelet should update `status` via `PATCH /status` so it doesn't overwrite spec changes (separation reduces conflicts).
- **finalizers**: used to ensure controllers complete cleanup before deletion (e.g., volumes detach).
- **resourceVersion**: used for watches and optimistic concurrency. If you try to `PUT` with an old resourceVersion, API server rejects (HTTP 409) — prevents lost updates.

---

7) Admission webhooks impact on pod creation

- **Mutating webhook** (e.g., sidecar injector) receives the object and can mutate it (adds sidecar, mounts, env). The mutated object is the one that gets persisted. This can affect scheduling (resource requests changed) and kubelet behavior.
- **Validating webhook** can reject creation (e.g., disallow privileged containers). If validation fails, Pod creation aborted — nothing written to etcd.
- Webhooks can run synchronously during creation (so they can delay the API response until they return).

---

8) Networking / CNI / kube-proxy

- **CNI**: kubelet calls CNI plugin to allocate an IP and configure networking in the Pod's netns (veth pair to node). The CNI result provides `podIP`.
- **kube-proxy**: creates Service routing rules in iptables/ipvs so Service IP -> Pod IP translation works.
- **DNS**: kube-dns/CoreDNS watches Service & Endpoints and resolves service names to ClusterIP.

---

9) Storage / Volumes (CSI)

- If Pod requests a PersistentVolume:

    - Controller (in control plane) may create a volume via CSI controller (CreateVolume).
    - Attach/Detach controllers ensure volume is attached to the node (for block volumes). kubelet mounts the volume into the Pod filesystem via CSI node plugin (NodePublishVolume).
- Volumes might delay Pod starting until attach/mount completes.

---

10) Lifecycle hooks, init containers, and termination

- **Init containers** run sequentially before normal containers; they must succeed, otherwise Pod stuck in `Init` state.
- **PreStop hooks** are run on graceful termination.
- **Graceful termination**: `kubectl delete pod` sets `deletionTimestamp` and kubelet runs termination logic, sends SIGTERM to containers and waits for grace period before SIGKILL.
- The `finalizers` and `ownerReferences` ensure resources are cleaned up.

---

11) Pod states & conditions (where they appear in etcd)

- `phase` can be: `Pending`, `Running`, `Succeeded`, `Failed`, `Unknown`.
- `conditions`: `Ready`, `PodScheduled`, `Initialized`, `Unschedulable` etc. These are set by controllers (scheduler sets `PodScheduled`, kubelet sets `Ready` and container statuses).
- `containerStatuses` includes `state.waiting/running/terminated`, `lastState`, `restartCount`, image IDs.

---

12) Race conditions & how Kubernetes avoids them

- **Optimistic concurrency** via `resourceVersion`.
- **Status subresource** to avoid spec/status clobbers.
- **Admission webhooks** can block conflicting changes early.
- **Leader election** among controllers via Lease objects in etcd.

---

13) Debugging — commands, what to look for, and where things commonly fail

Useful commands:

```
kubectl get pods -A
kubectl describe pod <pod> -n <ns>      # events, scheduling decisions, reasons
kubectl get pod <p> -o yaml
kubectl get events -n <ns> --sort-by='.lastTimestamp'
kubectl logs <pod> -c <container>       # container logs
kubectl get endpoints <svc> -o yaml
kubectl get pods -o wide                 # node assignment + IP
kubectl get nodes -o wide                # capacity
kubectl get replicaset,deploy -n <ns>
# To watch what scheduler sees:
kubectl get pods --field-selector=status.phase=Pending -w
```

Where things commonly fail and what you'll see:

- **Failed authentication/authorization**: API returns 401/403; `kubectl` error shows permission denied.
- **Admission webhook failures**: `kubectl` shows webhook error; `kubectl describe pod` shows rejected reason.
- **Scheduling failures**: Pod stays `Pending` and `kubectl describe pod` shows "0/5 nodes available: ..." with reason (Insufficient cpu/memory, nodeSelector mismatch, taints).
- **Image pull errors**: `containerStatuses[].state.waiting.reason = ImagePullBackOff` — check image name, pullSecret, registry credentials.
- **CNI failure**: Pod stuck in `ContainerCreating`; `kubectl describe` shows events from CNI; kubelet logs and CNI plugin logs needed.
- **Volume attach/mount issues**: Pod stuck in `ContainerCreating` with attach/mount errors; check CSI controller and node plugin logs.
- **Kubelet reporting problems**: Node `NotReady`, `kubectl describe node` for conditions; `kubelet` logs show issues.
- **CrashLoopBackOff**: Check container logs and liveness probe configuration.

Advanced debug:

- Inspect etcd key (careful — requires credentials & can be dangerous): `etcdctl get /registry/pods/<ns>/<pod> --write-out=...`
- Watch API events: `kubectl get events -w` or use `kubectl get pod <pod> -w` to see live transitions.

---

14) Example trace — a realistic short trace (with API calls)

1. `kubectl apply -f pod.yaml` → `POST /api/v1/namespaces/default/pods` (body = Pod manifest)
2. API Server: authN → authZ → mutating webhook → validating webhook → write to etcd `/registry/pods/default/nginx-1`.
3. Response returned to `kubectl` (Pod created).
4. Scheduler: `GET /api/v1/pods?fieldSelector=spec.nodeName=` returns unscheduled pods (or watch event ADDED).
5. Scheduler decides `node-05` → `POST /api/v1/namespaces/default/pods/nginx-1/binding` (target `node-05`).

6. API Server writes binding → etcd updated.
7. Node `node-05` kubelet gets watch event MODIFIED → enqueues pod.
8. Kubelet: calls CNI to allocate IP → calls CRI `RunPodSandbox` → `CreateContainer` for init containers & main containers → `StartContainer`.
9. Kubelet patches status: `PATCH /api/v1/namespaces/default/pods/nginx-1/status` with `podIP` and `containerStatuses`.
10. Pod transitions to Running and Ready after probe passes. Endpoints updated for services.

---

15) Extras — scheduler plugins and extenders, ephemeral containers, and debug hooks

- **Scheduler framework** supports plugins (permit, preFilter, filter, score, reserve, permit, preBind, bind, postBind). Custom schedulers or extenders can participate.
- **Ephemeral containers**: can be added into an existing Pod for debugging (API subresource).
- **API aggregation**: extension API servers can add new APIs; API server aggregates them into one endpoint.

---

16) Security & RBAC notes

- The API server enforces RBAC for who can create Pods, and kubelet enforces whether a pod can run privileged containers (PSA).
- ServiceAccounts issue tokens; controllers create Secrets (e.g., for pulling images) which are mounted to a Pod.

---

17) Best practices related to pod creation & lifecycle

- Use Deployments/ReplicaSets/StatefulSets to manage pods (don't create raw Pods except for debugging).
- Define resource `requests` and `limits` to help scheduler place pods and avoid resource contention.
- Use readiness probes to avoid sending traffic to not-yet-ready pods.
- Use liveness probes to recover from unhealthy containers.
- Use proper affinity/anti-affinity and taints/tolerations for topology spread and isolation.
- Use `PodDisruptionBudget` to limit voluntary evictions.
