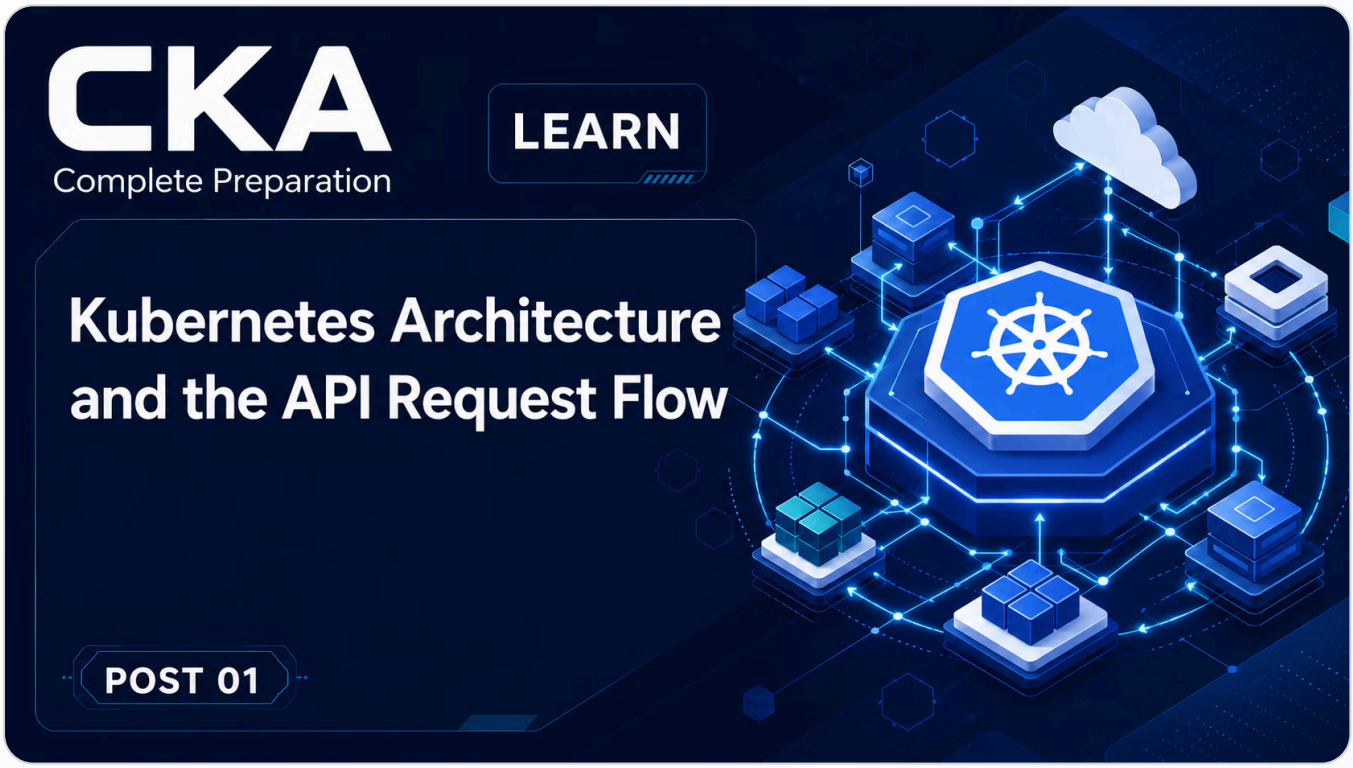
Omar Latrach

TECHNICAL ARTICLE

PUBLISHED · AUGUST 30, 2026

Kubernetes Architecture and the API Request Flow

Build the Kubernetes mental model every CKA candidate needs: trace a kubectl request through the API server, understand where cluster state lives, and see how stored intent becomes work on a node.



Most Kubernetes administration starts with a command and ends with a change somewhere in the cluster. The useful question is what connects those two points. If you can trace that path, unfamiliar resources and failures become much easier to reason about.

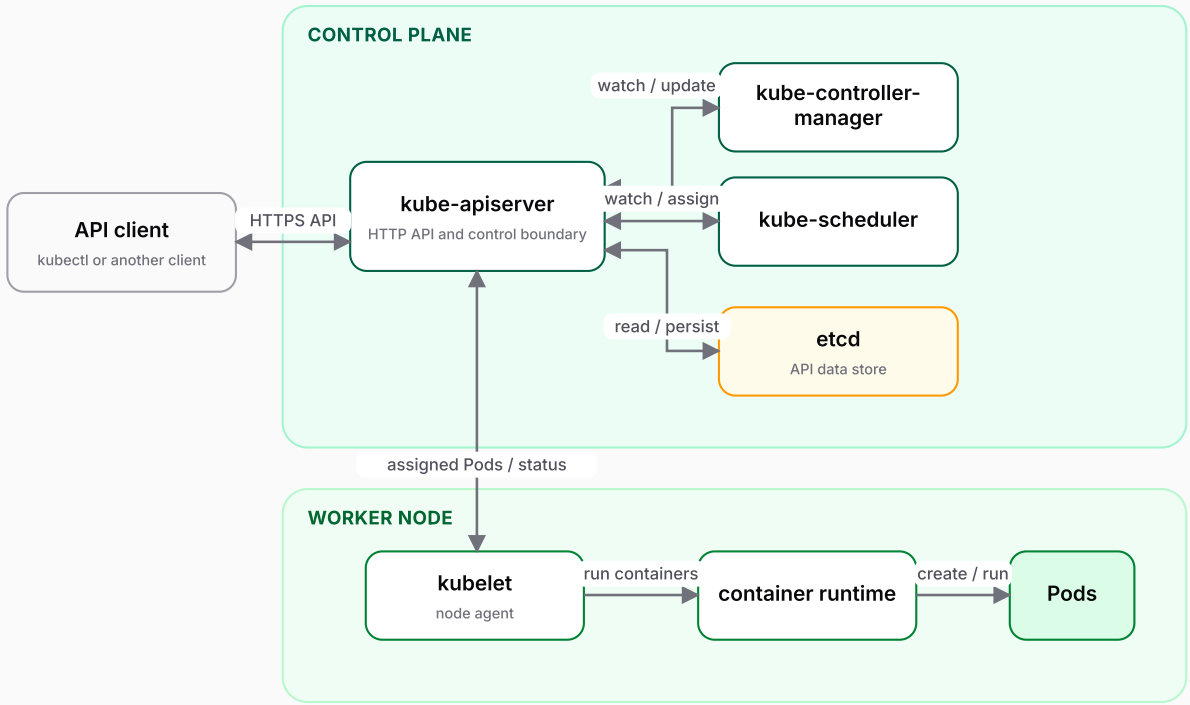
This lesson builds that foundation. We will look at the cluster as a control system, follow read and write requests through the Kubernetes API, and observe one normal request with kubectl. Later lessons will explore each component and resource type in more depth.

What you'll learn

- How the control plane and worker nodes divide responsibility.
- Why the API server is the front door for both administrators and Kubernetes components.
- How authentication, authorization, admission, validation, and persistence fit into a write request.
- Why a successful API response is the start of reconciliation, not proof that the requested outcome is already running.
- How to reveal the HTTP request behind kubectl and inspect fields supplied by the API server.

The mental model: front door, record, observers, actors

> The API server is the front door. etcd holds the authoritative record. Control loops observe that record and propose changes through the API. Node components turn assigned work into running containers.



Kubernetes coordination centers on the API server

Every component coordinates through kube-apiserver; controllers, the scheduler, and kubelet do not bypass it to access etcd.

The arrows matter more than the boxes. kubectl does not contact the scheduler, kubelet, or etcd. It sends an API request to kube-apiserver. In the normal Kubernetes architecture, the other components also coordinate by reading and writing API objects through that same server.

That shared API is what lets Kubernetes components remain loosely coupled. A scheduler does not need to command a kubelet directly. It records a scheduling decision through the API; the appropriate kubelet observes the assigned Pod and acts on it.

One cluster, two broad responsibilities

The control plane manages cluster state

- **kube-apiserver:** exposes the Kubernetes HTTP API. It authenticates and authorizes requests, runs applicable admission processing, validates objects, and provides the controlled path to stored cluster state.
- **etcd:** is the consistent key-value store that backs Kubernetes API data. It stores objects and control-plane state, not running containers. Administrators normally work through the Kubernetes API rather than editing etcd directly.
- **kube-controller-manager:** runs control loops. A controller watches relevant objects, compares desired state with observed state, and writes changes that move the cluster closer to the desired state.
- **kube-scheduler:** watches for Pods that do not yet have a node and selects a suitable node. It records that decision; it does not start the container.

Worker-node components run assigned workloads

- **kubelet:** is the node agent. It watches for Pods assigned to its node, works with the container runtime to make their containers run, and reports observed status through the API.
- **Container runtime:** pulls images and creates and runs containers when requested by kubelet.
- **kube-proxy, when used:** maintains node networking rules that help implement Services. Service networking belongs later in the series, so keep only the component's boundary in mind here.

This is intentionally a responsibility map, not a deployment map. Posts 15 and 16 examine how control-plane and node components actually run. High availability, static Pods, container interfaces, and networking implementation details come later.

Kubernetes is API-driven, not command-driven

kubectl is an API client. It reads connection and identity information from kubeconfig, builds an HTTP request for the selected resource, establishes a Transport Layer Security (TLS) connection to the API server, and sends the request. A client library or direct HTTP

client can use the same API.

Because the API is the common control surface, Kubernetes has a durable model of intent. A write does not mean “run these procedural steps once.” It means “accept this object as part of cluster state.” Controllers and node agents can continue acting on that state after the original kubectl process has exited.

The exact structure of apiVersion, kind, metadata, spec, and status is the subject of post 03. For now, treat a manifest as a typed request payload and the returned object as the server's representation of accepted state.

The read request flow

Consider kubectl get namespace default. At a useful CKA level, the request follows this path:

1. kubectl resolves the active cluster, user credentials, and context from kubeconfig.
2. The client establishes a TLS connection and verifies the API server's identity using its configured certificate authority.
3. **Authentication** determines who the requester is. A request that cannot authenticate is normally rejected with HTTP 401 Unauthorized.
4. **Authorization** determines whether that identity may perform the requested verb on the resource. An authenticated requester without permission is normally rejected with HTTP 403 Forbidden.
5. The API server returns the requested representation from cluster state, subject to its storage and caching behavior.

> Admission control does not run for ordinary read operations such as get, list, and watch. Admission is for requests that may change or connect to resources.

The write request flow

A create or update begins the same way, but it has additional gates before persistence:

1. **Receive and decode:** the API server identifies the API group, version, resource, namespace when applicable, name, and requested operation.
2. **Authenticate:** establish the requester's identity.
3. **Authorize:** decide whether that identity may perform this operation on this resource.
4. **Run admission:** applicable admission controllers may default or mutate the incoming object, and may reject it according to cluster policy. Validating admission checks the object after mutations.
5. **Validate:** the API server checks that the completed object is valid for the requested API resource.
6. **Persist and respond:** for a normal non-dry-run write, the accepted object is persisted in the API data store and the API server returns a response.

Because each gate happens before persistence, a rejected request does not become desired cluster state. Because persistence happens before asynchronous control loops finish their work, a successful create response confirms accepted state, not a running application.

Post 22 teaches Role-Based Access Control (RBAC) resources in detail. At this stage, remember the diagnostic distinction: 401 asks “who are you?”, while 403 asks “you are known, but may you do this?”

The URL describes the resource

kubectl resource syntax maps to REST-style API paths. You do not need to memorize every path, but recognizing the pattern makes verbose output readable.

Representative Kubernetes API paths · http

```
GET  /api/v1/namespaces
GET  /api/v1/namespaces/api-flow-demo
GET  /api/v1/namespaces/default/pods
POST /apis/apps/v1/namespaces/default/deployments
```

`/api/v1` is the core API group. Paths under `/apis/<group>/<version>` address named API groups such as `apps/v1`. Namespaced resources include the namespace in the path; cluster-scoped resources do not. Resource discovery, namespaces, and object fields are covered in posts 02 and 03.

Worked demonstration: watch a normal API request

This demonstration uses a disposable Namespace because it is a small API object and does not introduce workload behavior that belongs in later posts. Run it against a cluster where your current kubectl identity may create and delete namespaces.

1. Identify the API endpoint and make a direct API read

Use cluster-info when you need to confirm which control-plane endpoint kubectl can reach. Then ask the API server for its version endpoint through kubectl's authenticated connection.

```bash
kubectl cluster-info
kubectl get --raw=/version
```

The first command shows the configured control-plane address. The second returns a JSON version object. It is useful evidence that kubectl can complete TLS setup, authentication, authorization for this non-resource path, and an API read.

2. Reveal the HTTP request behind kubectl

Verbosity level 8 displays HTTP request contents. Use it when you need to confirm the method, URL, query parameters, or response status generated by kubectl.

```bash
kubectl get namespaces -v=8
```

The exact log format varies by kubectl version. Find the GET request to a path ending in /api/v1/namespaces and a successful response status. The final table is only kubectl's presentation of the API response. Verbose logs can contain operational details, so inspect them before sharing.

3. Generate the request payload locally

Client dry-run is useful for generating a manifest without contacting the API server. Because no request is sent, it cannot exercise server authorization, admission, or validation.

```bash
kubectl create namespace api-flow-demo \
  --dry-run=client \
  -o yaml > namespace.yaml
```

namespace.yaml · yaml

```yaml
apiVersion: v1
kind: Namespace
metadata:
  creationTimestamp: null
  name: api-flow-demo
spec: {}
status: {}
```

This file expresses a small desired object. The null timestamp is a clue that only the client has generated it; the API server has not persisted it or supplied server-managed metadata.

4. Compare server dry-run with a real write

Server dry-run sends the request to the API server and exercises server-side processing, but asks the server not to persist the object. It is therefore different from client dry-run.

```bash
kubectl create -f namespace.yaml --dry-run=server -o yaml

kubectl get namespace api-flow-demo
# Expected: NotFound, because the server dry-run was not persisted.
```

Now send the normal create request with verbose logging:

```bash
kubectl create -f namespace.yaml -v=8
```

Look for POST /api/v1/namespaces and the returned Namespace object. The final namespace/api-flow-demo created message means the API server accepted and persisted the object.

5. Read back the server's representation

```bash
kubectl get namespace api-flow-demo -o yaml
```

Relevant fields; values are cluster-specific · yaml

```yaml
apiVersion: v1
kind: Namespace
metadata:
  creationTimestamp: "<server timestamp>"
  name: api-flow-demo
  resourceVersion: "<opaque value>"
  uid: <server-assigned UID>
spec:
  finalizers:
    - kubernetes
status:
  phase: Active
```

The important observation is not the exact values. The server has assigned identity and version metadata and returned current status. Admission or other server behavior may also add fields. Post 03 explains metadata, spec, status, and resourceVersion properly.

6. Remove the demonstration object

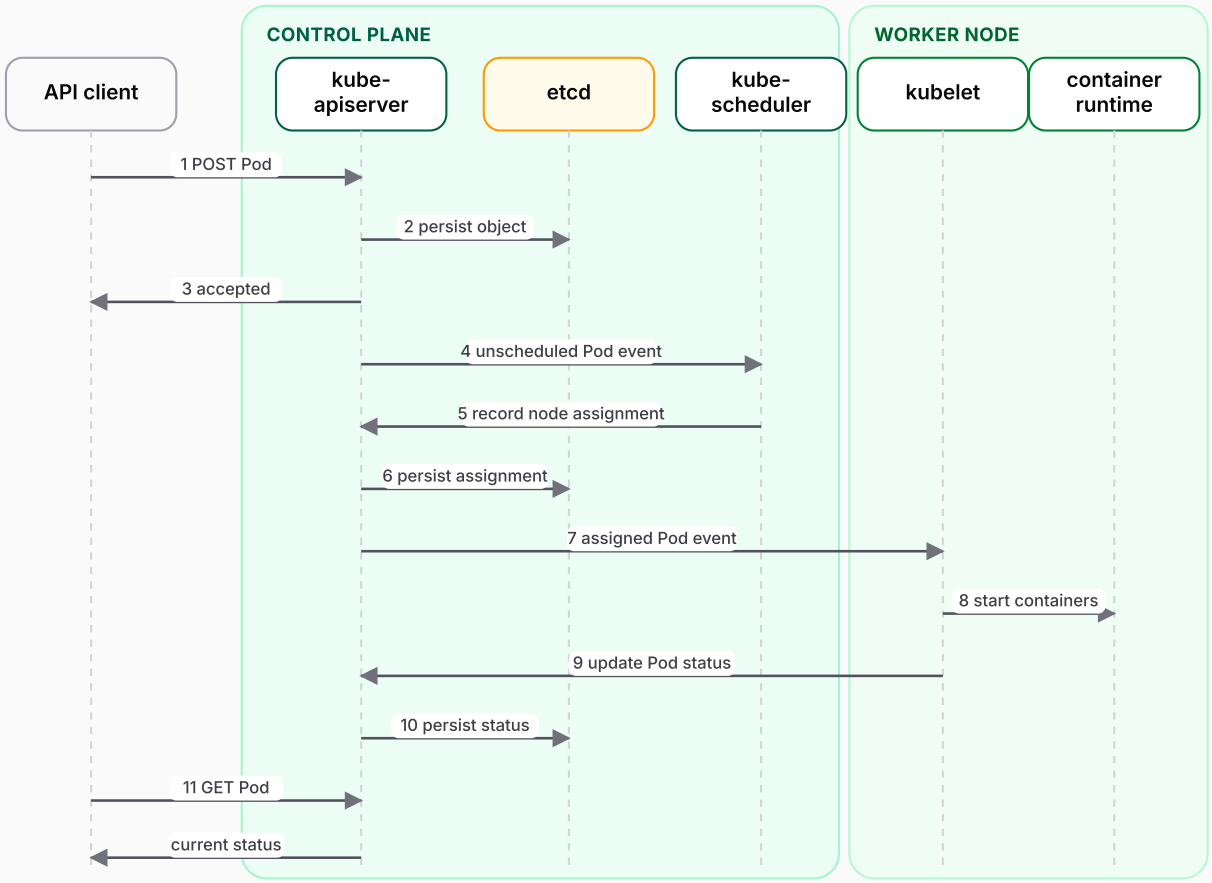
bash

```bash
kubectl delete -f namespace.yaml
```

Deletion is another API write. The command requests deletion; Kubernetes then handles the resource's deletion lifecycle. Namespace deletion details are outside this lesson.

After storage: how intent becomes action

Creating a Namespace mostly changes API state. Workload objects make the next half of the architecture easier to see. At a deliberately high level, a Pod request follows this causal chain:



From an accepted Pod request to reported status

The scheduler and kubelet never hand the Pod directly to each other; the Pod object in the API is their coordination point.

The scheduler does not run the container. The API server does not run the container. The kubelet and container runtime do that on the selected node.

Controllers use the same pattern for resources they manage: list or watch relevant objects, compare desired and observed state, and write an adjustment through the API. If the world changes again, reconciliation runs again. That repeated observation-and-correction loop is the core of Kubernetes self-healing.

Pods, Deployments, scheduling rules, and component internals each have later lessons. The point here is the coordination model: components react to shared API state rather than passing a single imperative command down a chain.

Use the flow to locate failures

When an operation fails, first decide whether the request failed or the requested state was accepted but has not produced the expected outcome.

- **Connection or TLS error:** the client did not successfully enter the API request pipeline. Check endpoint, context, network reachability, and certificate trust.
- **401 Unauthorized:** authentication did not establish an accepted identity.
- **403 Forbidden:** the identity is known but is not authorized for this operation.
- **Admission or validation rejection:** the write was refused before persistence. Read the returned Status message; changing node state cannot fix an object that was never accepted.
- **Object exists, outcome is incomplete:** the API write succeeded. Move your attention to reconciliation, scheduling, node agents, and reported status rather than repeatedly resubmitting the same object.

Later Learn posts provide the commands for each layer, and post 36 builds the full observability and troubleshooting workflow. For now, this boundary prevents a common mistake: treating every failed outcome as a failed kubectl request.

Official documentation

- Kubernetes Components
- Controlling Access to the Kubernetes API
- Kubernetes API Concepts
- kubectl Quick Reference