

Kubernetes Advanced

Visual handwritten study guide - Tier 3

Understand

Clear, compact explanations

Visualize

Diagrams that show the moving parts

Practice

Readable commands and code

Remember

Highlighted revision points

59 topics - each topic begins on a fresh page

A digital asset by Sooraj | soorajdev.cloud

Course Map

Scan the complete learning path before you begin.

Part 1 - API Server and Storage Internals

- | | | | |
|---|--|---|---|
| 1 | Course Introduction: From Operator to Platform Engineer | 2 | API Machinery: Scheme, Versions, Conversion and the Request Chain |
| 3 | Watches, resourceVersion and the Watch Cache | 4 | Extending the API: CRDs, Aggregation and Choosing Between Them |
| 5 | Admission in Depth: Webhooks, Mutating Policy and Failure Design | 6 | etcd at Scale: Performance, Maintenance and Disaster Handling |

Part 2 - Controllers and Extension Development

- | | | | |
|----|---|----|--|
| 7 | client-go: Informers, Listers, Work Queues and Rate Limiting | 8 | controller-runtime and kubebuilder: Production Operator Patterns |
| 9 | Designing APIs That Last: Conventions, Versioning and Compatibility | 10 | Testing Controllers and Operators |
| 11 | Building Admission and Conversion Webhooks in Go | 12 | The Scheduler Framework: Plugins, Profiles and Custom Schedulers |

Part 3 - Nodes, Runtimes and the Kernel

- | | | | |
|----|---|----|---|
| 13 | Kubelet Internals: Sync Loop, PLEG, Status and Pod Lifecycle | 14 | Container Runtimes: CRI, containerd, OCI and the Execution Stack |
| 15 | cgroups v2 and Node Resource Management: CPU, Memory, NUMA | 16 | Devices and Accelerators: Device Plugins, CDI and Dynamic Resource Allocation |
| 17 | Node Pressure, Eviction, Swap and In-Place Pod Resize Internals | 18 | Linux Tuning and Low-Level Diagnosis for Kubernetes Nodes |

Part 4 - Advanced Networking

- | | | | |
|----|--|----|--|
| 19 | CNI Internals: Pod Networking from Namespace to Wire | 20 | eBPF Networking with Cilium: Maps, Identities and kube-proxy Replacement |
| 21 | The Service Dataplane: iptables, nftables, IPVS and Conntrack at Scale | 22 | BGP, Bare-Metal Load Balancing and Edge Routing |
| 23 | IPv6, Dual-Stack and Address Planning | 24 | Multi-Cluster Networking and Service Discovery |
| 25 | Service Mesh Internals: Envoy, xDS, Ambient and Mutual TLS | 26 | Network Troubleshooting Like a Packet Detective |

Part 5 - Scale, Performance and Reliability Engineering

- | | | | |
|----|--|----|---|
| 27 | Scalability: SLOs, Limits and What Actually Breaks First | 28 | API Server Performance: Flow Control, Caching and Client Discipline |
| 29 | Scheduler and Controller Throughput at Scale | 30 | Load Testing and Simulation: KWOK, kube-burner and clusterloader2 |
| 31 | Profiling and Tracing the Control Plane | 32 | Cost, Efficiency and Capacity at Fleet Scale |
| 33 | Reliability Architecture: Failure Domains, Cells and Disaster Recovery | | |

Part 6 - Fleets and Platform Engineering

- | | | | |
|----|--|----|---|
| 34 | Cluster API: Declarative Cluster Lifecycle | 35 | Fleet GitOps: Multi-Cluster Delivery with Argo CD and Flux |
| 36 | Upgrading a Fleet: Strategy, Automation and Risk Management | 37 | Advanced Multi-Tenancy: Virtual Clusters, Hierarchies and Control Plane Isolation |
| 38 | Platform Engineering: Golden Paths, Internal Developer Platforms and Backstage | 39 | Crossplane and Composition: Building Infrastructure APIs |
| 40 | Policy, Governance and Compliance as Platform Features | | |

Part 7 - Advanced Security

- | | | | |
|----|--|----|---|
| 41 | Threat Modelling Kubernetes: Attack Paths and Defences | 42 | Workload Identity: Tokens, SPIFFE and SPIRE |
| 43 | Runtime Security: seccomp, LSMs, User Namespaces and eBPF Detection | 44 | Sandboxing and Confidential Computing: gVisor, Kata, Firecracker and TEEs |
| 45 | Software Supply Chain Security in Depth: SLSA, Attestations and Verification | 46 | Secrets, Encryption and the Cluster PKI |
| 47 | Hardening, Forensics and Red-Team Exercises | | |

Part 8 - Special Workloads, Projects and Reference

- | | | | |
|------------|--|------------|--|
| 48 | Stateful Systems at Scale: Local Storage, Rook-Ceph and Database Operations | 49 | Batch, HPC and AI Workloads: Kueue, JobSet, Volcano and GPU Scheduling |
| 50 | Observability at Scale: Sharding Prometheus, Thanos and Mimir, Cardinality Control | 51 | Edge, k3s and Constrained Environments |
| PROJECT 1 | Project 1: Build and Ship a Production Operator | PROJECT 2 | Project 2: A Fleet from Code with Cluster API and GitOps |
| PROJECT 3 | Project 3: Find the Limits of a Cluster | PROJECT 4 | Project 4: Attack and Defend a Cluster |
| 56 | Review Questions | APPENDIX A | Internals Cheat Sheet: Components, Ports, Metrics and Flags |
| APPENDIX B | Troubleshooting at Scale: Symptoms, Causes and Evidence | APPENDIX C | Glossary, Reading List and Where to Go Next |

Course Introduction: From Operator to Platform Engineer

What this volume assumes, how it differs from the earlier two, and the lab setup it needs.

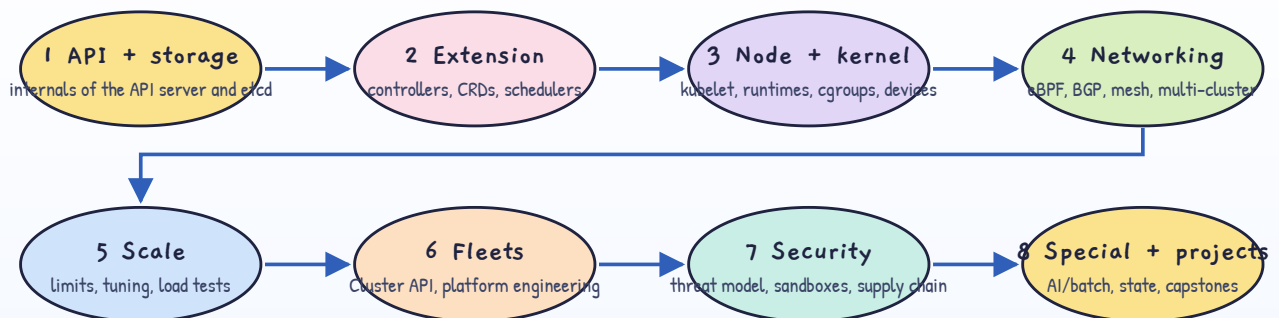
The Beginner volume taught you to **use** Kubernetes. The Intermediate volume taught you to **operate** it. This Advanced volume teaches you to **understand and extend** it: to read what the API server is doing, write controllers and schedulers, tune a cluster for scale, run fleets of clusters, and defend all of it against a capable attacker.

Watch out: This volume assumes the Intermediate material cold: control plane roles, etcd backup and restore, server-side apply, scheduling constraints, CNI and Gateway API basics, RBAC and policy, Helm and GitOps. If you cannot explain a reconcile loop or restore etcd from a snapshot, go back first. Nothing here re-teaches them.

What you should be able to do before starting

Skill	Self-test
Control plane	Draw the path of a <code>kubectl apply</code> through the API server to etcd and back out to a kubelet
Controllers	Explain level-triggered reconciliation and why controllers must be idempotent
Operations	Upgrade a kubeadm cluster one minor version and verify it
Linux	Explain namespaces, cgroups v2, and read <code>ss</code> , <code>ip</code> , <code>strace</code> output at a basic level
A systems language	Read Go (most of the ecosystem) or at least follow a 100-line Go program
Networking	Explain what a CNI plugin does and what happens to a packet between two pods on different nodes

DIAGRAM The eight parts of this volume



Each part ends with labs; Part 8 contains four capstone projects.

How this volume teaches

- ✓ **Mechanism first.** You learn the data structure or protocol, then the command that exposes it. If you know that a watch is a resumable stream keyed by `resourceVersion`, a thousand symptoms become predictable.
- ✓ **Source pointers.** Topics name the relevant upstream directories (`staging/src/k8s.io/apiserver`, `pkg/kubelet`, `pkg/scheduler`) and Kubernetes Enhancement Proposals (KEPs) so you can read the real thing. Read code, not blog posts, whenever behaviour is in doubt.
- ✓ **Trade-offs and failure modes.** Advanced work is choosing between imperfect options. Each topic says what a design costs.

- ✓ **Version facts** are written for Kubernetes 1.37 (October 2026). Feature gates and defaults change between releases; always check the release notes and `kubectl api-resources` /feature-gate tables for your version.
- ✓ **Code is illustrative**. Go and config listings show shape and idioms; they are trimmed to fit a page and are not drop-in production code.

The lab environment

Use real virtual machines or a nested-virtualisation host where possible. Many topics (kernel tuning, runtimes, eBPF, device management) cannot be demonstrated honestly in a container-in-container kind cluster. Plan for the following.

Lab	Specification	Used for
Core cluster	3 control-plane + 3 worker VMs, 4 vCPU and 8 GiB each, Linux 6.x kernel with cgroup v2	Parts 1, 3, 4, 5, 7
kind or k3d fleet	5 to 10 small clusters on one host	Part 6 (Cluster API, fleet GitOps)
Dev loop	Go toolchain, kubebuilder, envtest binaries, <code>kind</code> , <code>ko</code> or Docker	Part 2 and capstone projects
Observability	Prometheus, Grafana, Tempo, pprof access, <code>bpfftrace</code> , <code>perf</code>	Parts 3, 5
Scale simulation	KWOK (fake nodes), kube-burner, clusterloader2	Part 5
Attacker box	A separate namespace or cluster with attacker tooling	Part 7

```

● ● ● BASH SNIPPET 7 lines
# toolchain checklist
$ go version # a current Go release
$ kind version ; kubectl version --client ; helm version --short
$ kubebuilder version # scaffolding for Part 2
$ setup-envtest list # API server + etcd binaries for tests
$ bpfftrace --version ; perf --version ; crictl --version
$ uname -r && stat -fc %T /sys/fs/cgroup # expect cgroup2fs

```

Reading source code efficiently

- 1 Clone `kubernetes/kubernetes` and learn the layout: `cmd/` (binaries), `pkg/` (kubelet, scheduler, controllers), `staging/src/k8s.io/` (published libraries: `client-go`, `apiserver`, `api`, `apimachinery`).
- 2 Start from a metric, log line or error string: grep it, then read the call chain upward. This beats reading top-down.
- 3 Use `git log -S` and `git blame` to find the PR and KEP that introduced a behaviour; the PR discussion usually states the reason.
- 4 Read tests: `*_test.go` and `test/integration` show intended behaviour better than comments.
- 5 Run a component locally with `-v=6` or higher logging in your lab and watch what it does for one request.

Treat everything about alpha and beta features as provisional. Production decisions should rest on **GA** behaviour plus **beta features you have tested at your scale**. KEP status, not blog enthusiasm, is the source of truth.

Q Why does this volume stress reading source and KEPs?

A Advanced operators diagnose behaviour that documentation does not cover. The source and the design proposals are the only complete statement of intent.

Q Why is a kind cluster insufficient for some labs?

A It shares the host kernel and hides node-level behaviour such as runtimes, device plugins, cgroup managers and kernel tuning.

Key takeaways

- Advanced work is understanding mechanisms, extending the system and operating it at scale.
- Prepare a VM-based lab with a modern kernel, plus fleet and scale-simulation tooling.
- Read upstream code and KEPs; treat alpha and beta features as provisional.

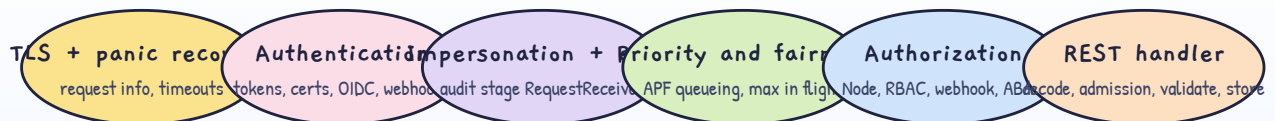
API Machinery: Scheme, Versions, Conversion and the Request Chain

How the API server turns bytes on a socket into a typed, versioned, validated object, and back again.

The Kubernetes API server is mostly a **generic framework** (k8s.io/apiserver) with Kubernetes types registered into it. Understanding the framework explains why every resource, built-in or custom, behaves consistently: same verbs, same watch semantics, same status handling, same server-side apply.

The handler chain

DIAGRAM A request from socket to storage



Filters wrap each other; the first to reject ends the request. Order matters for debugging.

Stage	What it does	Failure looks like
Request info	Parses verb, API group, version, resource, namespace, subresource from the path	404 for an unknown path
Authentication	First authenticator that succeeds wins; produces user, groups, extra	401; audit shows <code>system:anonymous</code>
Audit	Records stages (<code>RequestReceived</code> , <code>ResponseStarted</code> , <code>ResponseComplete</code> , <code>Panic</code>)	Missing events if the policy omits the resource
APF	Classifies via FlowSchema into a PriorityLevel, queues fairly, rejects with 429 when full	429 with <code>Retry-After</code> ; <code>apiserver_flowcontrol_*</code> metrics
Authorization	Node, RBAC, webhook... chained; any allow wins; no opinion falls through	403 with the reason string
Admission (mutating then validating)	Plugins and webhooks may change or reject the object	Rejection message names the plugin or webhook
Validation and strategy	Resource strategy defaults, validates, prepares for update	422 Invalid with field errors
Storage	Encode, optionally encrypt, write to etcd with preconditions	409 Conflict on stale <code>resourceVersion</code>

Scheme, types and codecs

A **Scheme** (`runtime.Scheme`) is a registry mapping Go types to **GroupVersionKinds** and holding conversion and defaulting functions. A **codec** serialises objects to JSON, YAML or Protobuf and negotiates the wire version.

Concept	Meaning
Internal version	A hub representation the server uses in memory; each external version converts to and from it
External versions	<code>v1</code> , <code>v1beta1</code> , <code>v2</code> : what clients see; each has its own Go types, defaults and conversions
Storage version	The version persisted in etcd for a resource; set per resource; can differ from the served version
Defaulting	Functions applied on decode and after conversion to fill omitted fields
Conversion	Generated (<code>conversion-gen</code>) functions between versions; CRDs use <code>None</code> or a conversion webhook
Protobuf	Built-in resources are stored and served as protobuf by default to save size and CPU; CRDs use JSON

```

OUTPUT 14 lines

// Sketch: how a type is registered (staging/src/k8s.io/api style)
var SchemeGroupVersion = schema.GroupVersion{Group: "apps", Version: "v1"}

func addKnownTypes(scheme *runtime.Scheme) error {
    scheme.AddKnownTypes(SchemeGroupVersion,
        &Deployment{}, &DeploymentList{},
        &StatefulSet{}, &StatefulSetList{},
    )
    metav1.AddToGroupVersion(scheme, SchemeGroupVersion) // adds ListOptions, WatchEvent, etc.
    return nil
}

// Decode path (simplified): bytes -> universal deserializer -> external typed object
// -> defaulting funcs -> convert to internal -> admission -> validate -> convert to storage version ->
// encode -> etcd

```

Seeing versions and storage in practice

```

BASH SNIPPET 7 lines

$ kubectl api-versions | head
$ kubectl api-resources -o wide | head -20      # verbs, short names, namespaced, kind
$ kubectl get --raw /openapi/v3 | jq '.paths | keys | length'
$ kubectl get --raw /apis/apps/v1 | jq '.resources[] | select(.name=="deployments") |
{name,verbs,storageVersionHash}'
$ kubectl get storageversions.internal.apiserver.k8s.io      # records the storage version per resource
(when the feature is on)
$ kubectl explain deployment.spec.strategy --api-version=apps/v1
$ kubectl get deploy web -o yaml --show-managed-fields | head -60

```

Content negotiation and the wire

Header or parameter	Effect
<code>Accept: application/json;as=Table;g=meta.k8s.io;v=v1</code>	Server-side printing: the server returns table columns (how <code>kubectl get</code> renders custom columns)
<code>Accept: application/vnd.kubernetes.protobuf</code>	Protobuf for built-in types; smaller and faster than JSON
<code>Accept: ...;as=PartialObjectMetadata</code>	Only metadata: ideal for controllers that need names and labels, not specs
<code>?limit=500&continue=...</code>	Chunked LIST with an opaque continue token; avoids a huge single response
<code>?resourceVersion=0</code>	Serve from the API server's watch cache , possibly stale but cheap
<code>?watch=true&allowWatchBookmarks=true</code>	Stream of ADDED, MODIFIED, DELETED and BOOKMARK events
<code>?fieldManager=x&force=true</code> (PATCH)	Server-side apply with ownership
<code>?dryRun=All</code>	Runs the entire chain except persistence

How many API groups does the server really serve?

The `kube-apiserver` binary is actually a chain of three delegating servers: the main `KubeAPIServer` (built-in resources), the `APIExtensions server` (CRDs), and the `aggregator` (`kube-aggregator`) in front, which proxies requests for registered `APIService` objects to extension API servers (metrics-server, custom APIs). Requests hit the aggregator first, which decides whether to proxy or delegate downward.

This is why a broken `v1beta1.metrics.k8s.io` `APIService` can slow namespace deletion and `kubectl api-resources`: discovery and garbage collection must contact every registered API server.

Debugging with verbose client and server logs

```
BASH SNIPPET 5 lines
$ kubectl get pods -v=8 2>&1 | head -40 # show requests, headers and response bodies (redact tokens before sharing)
$ kubectl get pods -v=6 # just the URLs and status codes
$ kubectl proxy --port=8001 &
$ curl -s localhost:8001/api/v1/namespaces/default/pods?limit=1 | jq '.metadata'
$ kubectl get --raw '/api/v1/namespaces/default/pods?watch=true&timeoutSeconds=5' | head -3
```

HANDS-ON LAB Lab 1: Follow one request through the chain

1. Enable audit logging with a `RequestResponse` policy for one namespace in the lab cluster.
2. Create a Deployment with `kubectl -v=8` and map each HTTP request `kubectl` makes (discovery, POST) to audit events.
3. Add a mutating webhook (or use a policy) that adds a label, repeat the create, and locate the mutation in the returned object versus the submitted one.
4. Use `?dryRun=All` and confirm nothing was persisted but admission still ran.
5. Capture a watch with `curl` and identify ADDED, MODIFIED and BOOKMARK events.