Here’s What’s Actually Going On with Kubernetes Administration

The complete Kubernetes administration guide in plain language - Free sample (Chapter 1)



Luis Felipe Ariza Vesga

Here's What's Actually Going On with Kubernetes Administration

Here's What's Actually Going On with Kubernetes Administration

The complete Kubernetes administration guide in plain language - Free sample (Chapter 1)

Luis Felipe Ariza Vesga

Here's What's Actually Going On with Kubernetes Administration
The complete Kubernetes administration guide in plain language - Free sample (*Chapter 1*)

Copyright © Luis Felipe Ariza Vesga.
All rights reserved.

The information in this book is distributed on an "as is" basis, without warranty. While every precaution has been taken in the preparation of this work, neither the author nor the publisher shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this book.

ISBN: 9798178782736

For the western black rhinoceros (Diceros bicornis longipes), which lived in the savannas of Cameroon, Chad, the Central African Republic and northeastern Nigeria, and held up that entire savanna just by grazing and walking through it. Poaching for its horns and farmers killing it to protect their crops wiped out 96 percent of its population between 1970 and 1992; the last confirmed trace was recorded in 2006, in Cameroon's Northern Province, and the IUCN declared it extinct in 2011. And for everyone who administers a cluster: the most important work is almost always the work no one notices until it is gone.

Contents

Conventions Used in This Book **1**
Typography and References 2
The Four Callout Boxes 2
Third-party credits and licenses 3

Preface **5**
What This Book Is About 5
What This Book Is Not 6
How This Book Is Organized 7
About the Commands and Code Examples 8
How to Contact Us 10
Acknowledgments 10

I Cluster Architecture, Installation and Configuration (25%) **13**

1 The foundations of the house: who does what in the cluster **15**
1.1 Dividing up the trades before laying a single brick 15
1.2 The single window: why everything goes through the kube-apiserver 17
1.3 The brains of the site: the ledger, the foreman, and the overseers 20
1.4 The trades in each area of the site: the resident worker, the mail carrier, and the contractor 23
1.5 Two houses instead of one: the extra cost that prevents ruin . . 25
1.6 A site with more than one head office: static Pods and high availability 26
1.7 Practice in the lab 28

1.8 Summary 28
Bibliography 29

Conventions Used in This Book

This book's small visual system, demonstrated live, not only described.

Before we start, half a page on how this book is written. It reads in two minutes and saves confusion later.

This book speaks to you with the respect owed to a colleague you work shoulder to shoulder with.

Every chapter opens with an everyday scene. That scene is not decoration or padding: it **is** the technical concept told another way, and it will come back at the end of the chapter, in the summary, explained with the proper words. If at any point a technical explanation does not land, go back to the opening scene: the answer is almost always sitting there in simpler form. And those scenes are not loose pieces: they form a single story, that of the estate whose house is built in Chapter 1 and is still being looked after in Chapter 14, so each chapter picks up what the earlier ones left behind.

This book is not out to make you laugh on purpose. What it does is translate technical language into everyday language, using real-life stories everyone has lived through, familiar and easy to recognize, used as analogies to quickly understand a Kubernetes concept that has traditionally been explained in straight technical language, which makes it harder to study. That includes facts such as `kubectl cordon` marking a node to receive nothing new while the Pods already running there carry on as if nothing happened, or a badly written `NetworkPolicy` leaving an entire namespace mute without a single alarm going off: if that gets a smile out of you, welcome, but the goal is never the joke, it is that you understand. That is why the analogy never replaces rigour: after it always comes the official definition, quoted word for word, with its link to the source.

Typography and References

Monospaced type is used for anything you would type or read on a screen: object names such as `Pod` and `Deployment`, YAML fields such as `restartPolicy`, commands such as `kubectl describe pod`, and paths such as `/etc/kubernetes/manifests`.

Bold marks the first appearance of an important term, and also the words that tend to appear unchanged in the wording of the exam's tasks. When you see a bold word inside a definition, it is because that precise word matters.

Angle brackets mark what you must replace: in `kubectl logs <pod-name>`, you type the real name and you do not type the brackets.

Quotations from the official documentation appear woven into the text rather than as separate blocks, and they always carry a link to the source the first time they are used in a chapter. That is intentional: you should be able to verify any claim in this book without searching for it, by clicking.

Every chapter closes with a **bibliography** of at least three sources, all official and all with links.

Figures are referenced in the text and carry an explanatory caption. If a figure sits near an explanation, it is because that explanation reads better alongside it.

The Four Callout Boxes

Throughout the book you will meet four kinds of box. Each means something different and adds something the prose did not say: a command to verify with, a shortcut for the exam terminal, a behaviour observed on a real cluster or a production risk. None of them warns you about trick questions, because the CKA has none: it is a hands-on exam that assigns you tasks and grades the final state of the cluster. What the boxes do is help you leave that state done right and checked. It pays to recognize them at a glance, because they do not all carry the same urgency.

 Note
Complementary information expanding on what you just read, usually with a way to check it yourself. It can be skipped without losing the thread. Example: the control plane components kubeadm installs are static Pods, defined as files in `/etc/kubernetes/manifests` and managed directly by the node's kubelet, without the kube-apiserver watching over them or the scheduler placing them.

 Tip
A shortcut for the exam terminal, a mnemonic or a faster way of doing something. Example: `kubectl create deployment golden-spoon --image=nginx --replicas=2 --dry-run=client -o yaml` writes the skeleton of the manifest in a second, without creating anything in the cluster; you redirect it to a file and edit only what the task asks for.

 Warning
A frequent mistake or a behaviour that surprises people on a real cluster, the kind that leaves a task half solved. If you are only going to read one kind of box before sitting the exam, make it these. Example: `kubectl cordon` marks the node as unschedulable, but it does not remove the Pods that were already running on it; emptying it takes `kubectl drain`.

 Caution
Something that can cause real damage on a production cluster: data loss, an outage or a security hole. These are few and they are serious. Example: a NetworkPolicy with `podSelector: {},policyTypes: [Ingress]` and no ingress rules leaves every Pod in your namespace without inbound traffic, provided the cluster's network plugin enforces NetworkPolicy.

Third-party credits and licenses

This book reproduces and adapts excerpts from the official Kubernetes documentation (https://kubernetes.io/docs/), published by The Kubernetes Authors under the Creative Commons Attribution

4.0 International (CC BY 4.0) license. The license text is available at https://creativecommons.org/licenses/by/4.0/. The changes from the original are of two kinds: the translation into Spanish and the everyday story that accompanies each concept; the official definition always appears separately, unaltered, with its link to the source.

Definitions quoted verbatim appear in quotation marks and link to the page they come from. Any adaptation or paraphrase of that text is this book's own and has not been reviewed or endorsed by The Kubernetes Authors. The links are there so you can check every claim at its origin, and if an official page changes after the book is published, the current source is the one to consult.

Kubernetes is a registered trademark of The Linux Foundation. The names of the certifications cited belong to the Linux Foundation and the Cloud Native Computing Foundation. This book is independent study material, not affiliated with, endorsed by, or sponsored by those organizations. The names of the other projects and products that appear throughout the book, such as kind, etcd or CoreDNS, belong to their respective owners and are mentioned only to identify the software used for practice.

Preface

How this book is organized, the voice it explains the CKA exam in, and what to expect from the five parts ahead.

This book grows out of the same question that started the whole "Here's What's Actually Going On" series: what if the exam of the moment were explained in the voice we use to tell what happens on a country estate or in any neighborhood? The Kubernetes administration certification exam, the CKA (Certified Kubernetes Administrator), gets painted as a dry exam: kube-apiserver, etcd, kubelet, RBAC (Role-Based Access Control), taints and tolerations, words that sound like a server room and nothing else. Yet underneath that vocabulary, administering a Kubernetes cluster looks an awful lot like running a large estate: if we want to build a new house, we have to lay its foundations, hand out the keys, hire the trades, organize the shifts at The Golden Spoon restaurant, watch who comes in through the street door and, above all, know what to do when something breaks and nobody knows why. That thread runs through all fourteen chapters: behind every object in the Kubernetes API (Application Programming Interface) there is a household chore you already understand by heart.

What This Book Is About

This is a preparation book for the CKA exam, Certified Kubernetes Administrator, from the Cloud Native Computing Foundation (CNCF) and the Linux Foundation. It follows curriculum version v1.35, organized into five parts that follow the exam's five current official domains: Cluster Architecture, Installation and Configuration (25%), Workloads and Scheduling (15%), Services and Networking (20%), Storage (10%) and Troubleshooting (30%). There are fourteen chapters, each one opening with an everyday scene that

turns into the precise technical explanation of a real Kubernetes mechanism, and closing with a summary that returns to that same scene. Each of the five parts has its own lab, run on an actual kind cluster with kubectl and kubeadm, not a filler exercise. The goal is not for you to memorize five percentages, but to finish the book understanding why the exam gives them that weight: Troubleshooting is worth 30%, more than any other domain, because an administrator who knows how to install a cluster but cannot tell why a Pod ended up in CrashLoopBackOff is no use the day something breaks for good.

A note about the version, because it is easy to get confused here. Why v1.35, if Kubernetes is already at 1.37? Because this book follows the exam's official curriculum, not the latest version of Kubernetes. As this edition was written, the curriculum published by the CNCF is v1.35, and the Linux Foundation states that the exam runs on Kubernetes v1.35. Kubernetes 1.37 came out on August 26, 2026, and the Linux Foundation aligns the exam with the most recent minor version between four and eight weeks after each release. The labs pin Kubernetes v1.36, which is within the supported versions. Before you schedule your exam, check the official CKA page and the github.com/cncf/curriculum repository: if the curriculum has already moved to a new version, that page overrules this book.

What This Book Is Not

This book is not an exhaustive reference manual for every field, every flag or every object in the Kubernetes API; for that there is the official documentation at kubernetes.io/docs, which is cited in every chapter. Nor is it a Linux course from scratch, or a course on containers or on Docker: it assumes you already know how to read a YAML file, write a Dockerfile, build images, get around a terminal and recognize what a container is without having it explained here. No specific Kubernetes distribution or particular cloud provider gets a chapter of its own: the book stays with what kubeadm installs and what the CKA exam assesses, on a generic cluster, not with how to configure this or that provider's managed service. And it definitely does not reproduce questions from real exams or from simulators like killer.sh: that would violate both ethics and the exam's confidentiality agreement, and it

would also be bad business, because memorizing one task does not help you the day the same task arrives with a different namespace name.

If you are only just starting out with Kubernetes, begin with the previous book in this series: **Here's What's Actually Going On with Kubernetes**, *The complete Kubernetes guide in plain language*, already published on Amazon. That book explains from scratch what a container is, what a Pod is, how a cluster is put together and why the system behaves the way it does, using real-life situations and everyday analogies, and it covers the KCNA (Kubernetes and Cloud Native Associate) exam syllabus. This book picks up where that one leaves off: it takes those concepts for granted and devotes itself to administering them. Read in that order, the two are much easier to follow.

How This Book Is Organized

The book follows the order of the exam's five official domains, grouped into five parts with their declared weights, and it tells a single story from start to finish: that of an estate where you build, hand out keys, hire, serve and look after things. In the first part (cluster architecture) the main house and its control house go up in Chapter 1, the keys are handed out with RBAC in Chapter 2, and Chapter 3 hires the crew that installs, upgrades and backs everything up with kubeadm, Helm, Kustomize, CRDs (Custom Resource Definitions) and operators. In the second part (workloads and scheduling) The Golden Spoon restaurant opens its doors: Chapter 4 organizes its shifts with Deployments, rollouts and autoscaling, and in Chapter 5 every employee receives a recipe, a secret ingredient and a place to work through ConfigMaps, Secrets and the scheduler's rules. The third part (services and networking) heads out into the neighborhood: the gossip that lets you find every neighbor even though people move house very often (Chapter 6), Justo Segura, the watchman who decides which traffic gets through (Chapter 7), and the street door where visitors come in through Ingress and the Gateway API (Chapter 8). The fourth part (storage) goes down to the storeroom: the shared depot with its volumes and access modes in Chapter 9, and the space that is reserved and grows on demand with StorageClasses in Chapter 10. The fifth and last part, the heaviest on the exam at 30%, is devoted to troubleshooting through four familiar figures: the family doctor and his method (Chapter

11), the mechanic who checks the control plane's engine (Chapter 12), the fuel gauge of resource usage (Chapter 13) and the phone when the call will not connect because the network fails (Chapter 14).

So that the story never gets tangled, the base analogy is fixed for the whole book:

Kubernetes	On the estate
Cluster	The whole estate
Namespace	One house on the estate: the main house, and later the children's and grandchildren's summer houses
Control plane	The control house, the site's head office
Worker node	A work front, a crew that can work on any house
Pod	Whoever lives and works inside: residents, workers, cooks

Within each part, the chapters move from the most fundamental to the most specific, and each one opens by recalling what the earlier ones built. The glossary appendix gathers the book's key terms, with this same analogy, for review before the exam, and each part's lab assumes you have already read that part's chapters, not those of later parts.

About the Commands and Code Examples

Every YAML and kubectl example in this book is real and verified against the official documentation at kubernetes.io/docs, with the fields the Kubernetes API expects and the resource names defined by the cluster's own OpenAPI schema, not pseudocode invented to look pretty. When an example uses kubectl, the book tells you the precise verb (get, describe, apply, rollout) and the resource it acts on, because mixing up `kubectl delete pod` and `kubectl delete deployment` is a common exam mistake that

deletes different things. When a behavior depends on the installed version, the book teaches you to look it up with `kubectl version` and in the release notes at kubernetes.io, instead of pinning a number that goes stale the day you read this. Only the labs pin a specific cluster version, by kind image digest, because a lab does need to be reproducible from start to finish, with no surprises from a new version that changed a field underneath you.

All the lab code is published at https://github.com/lfarizav/lo-que-pasa-es-que-administrar-kubernetes. The repository has two folders: `es/` for the Spanish edition and `en/` for the English edition. Inside each one there is one lab per part, from `lab-part-01` to `lab-part-05`, with its commands, its checks and the versions it was tested with, ready to clone and run in a Linux terminal:

```
git clone https://github.com/lfarizav/lo-que-pasa-\
es-que-administrar-kubernetes.git
cd lo-que-pasa-es-que-administrar-kubernetes/en
```

Every command in those labs was executed on a cluster before publication, and each lab brings up its own isolated kind cluster without touching your everyday kubeconfig, so never run one against a production cluster. They resemble what you will be asked to do, because CKA is not a multiple choice exam: according to the official certification page, it is a two hour hands-on test at the command line that assigns tasks and grades the state you leave in the cluster. They are still not a simulator and do not reproduce exam questions; they turn each claim in this book into something you watched happen in your own terminal.

Every chapter closes, right before its summary, with a **Practice in the lab** section that tells you which lab belongs to it and which Kubernetes topics you will work on there; the steps, commands and checks live only in the repository. If a command in the book does not work for you, check that repository first: when the official documentation changes, the fix lands there before the next printed edition.

How to Contact Us

This book is part of Luis Felipe Ariza Vesga's catalog. If you find an example that does not apply the way the book says, a field that was renamed in a recent Kubernetes version, a scheduler behavior that was updated after this chapter was written, or a claim that does not match the current official documentation, that feedback is welcome and taken seriously: a technical book that does not get corrected stops being trustworthy. Kubernetes is a project that evolves fast, with new versions every few months and API objects that graduate from beta to stable without warning, so a book on the subject has to accept that no edition is the last word, and that a correction made in time is worth more than a promise that none will ever be needed. This book would rather be corrected than pretty, as a cluster administrator would rather have an annoying liveness probe alert than a comfortable silence that hides a Pod that has already died.

Acknowledgments

This book did not invent anything from scratch, and that is precisely what it wants to give thanks for. Kubernetes exists because thousands of people decided not to reinvent the wheel each on their own, but to build a single wheel together, in plain sight.

To the Cloud Native Computing Foundation (CNCF) and the Linux Foundation, for giving Kubernetes and the projects around it a neutral home, for publishing the CKA exam curriculum openly, and for sustaining the certification this book prepares you for.

To the communities and companies that donated whole projects to the CNCF. Handing an entire project to a foundation, with its code, its name and its governance, instead of keeping it as a private advantage, is what lets an administrator anywhere in the world use the same tools without asking anyone's permission. Every chapter of this book rests on one of those donations.

To the maintainers of Kubernetes, kubeadm, etcd, CoreDNS, Helm, Kustomize, the Gateway API and the network and storage plugins that appear

in these pages. Reviewing other people's code, answering issues at midnight and saying no to a poorly thought-out feature is work that almost nobody sees and that everybody depends on.

To all the contributors: to those who write code, but also to those who write and translate the documentation at kubernetes.io, report bugs, test beta releases and answer questions in the forums. The official documentation this book cites in every chapter exists thanks to them, open and free for anyone.

To the local open source communities: to the members of Cloud Native Colombia, KCD Colombia (Kubernetes Community Days) and the Colombia OpenInfra User Group. They organize meetups, workshops and events to promote open source without charging for it, and they often pay out of their own pockets for the snacks, the transport or the venue so that each gathering can happen. A good part of what is told here was learned at those meetings.

And finally, thanks to the western black rhinoceros, whose story lends the cover its hardest lesson: that the work that holds up a whole system is rarely noticed while it exists, and that when that quiet work disappears, what falls is not a single piece but the entire savanna.

PART I

Cluster Architecture, Installation and Configuration (25%)

1

The foundations of the house: who does what in the cluster

Why a Kubernetes cluster is a lot like raising a house from the foundations up, and who does which job on that building site.

1.1 Dividing up the trades before laying a single brick

Imagine you finally decide to build the house on the *finca*, the family country estate, that you have been promising yourself for years. If you read the previous book in this series, *Here's What's Actually Going On with Kubernetes*, you already know the land from the fence line; now it is time to put on the hard hat and run the site from the inside. You don't start with the roof, or with picking the colour of the front wall: you start with the foundations. If the foundations are wrong, everything else comes down sooner or later, and you know it because you watched your neighbour's house fall, with half a lifetime of savings inside it.

A serious building site divides up the trades, and everyone sticks to their own job. There is a foreman, who decides which area of the site each crew works in. There are overseers, who walk the grounds checking that everything still matches the plan. There is a ledger, where every decision gets written down

so that nobody forgets it or makes it up later. And in each area there is a resident worker who receives the orders and calls in a trusted contractor: that contractor is the one who lays brick on brick, raises walls and columns, and runs the water, power and gas lines.

On the shelf of the head office, on top of the rolled-up plans, the foreman keeps a wooden figure with its paint worn off: a big, solid animal with two horns, a long one in front and a short one behind. Nobody knows where it came from and nobody asks. The one thing he does say, in passing and without turning around, whenever someone is in a hurry to see walls go up, is this: that the quiet work is what holds the site together, and that nobody notices that work until it is missing.

Here's what's going on: a Kubernetes cluster is organized with that same division of trades. The whole estate is the cluster. Each house on the estate is a namespace: today we raise the main house, and over the years the summer houses of the children and the grandchildren will follow. The people who live and work inside those houses are the Pods, with their containers. And there is one difference worth seeing from the very first page: before the main house, Kubernetes asks you to raise a second building, the control house, which serves as the site's head office.

That head office is the control plane, and four trades work there that never get mixed up with each other:

- the kube-apiserver, the single window through which every instruction enters;
- etcd, the site ledger where the state of everything is written down;
- the kube-scheduler, the foreman, who decides in which area of the site (which node) each new job gets done;
- the kube-controller-manager, which brings together the overseers who keep checking, without pause, that the site still looks like the plan.

In each area of the site, that is, on each worker node, lives the kubelet, the resident worker who receives the orders and sees that they get carried out. Next to it are the kube-proxy, who makes sure the mail reaches the right place, and the contractor, the container runtime, who lays every brick. An area of the site is not tied to a single house: its crew can work on the main house today and on the grandchildren's tomorrow.

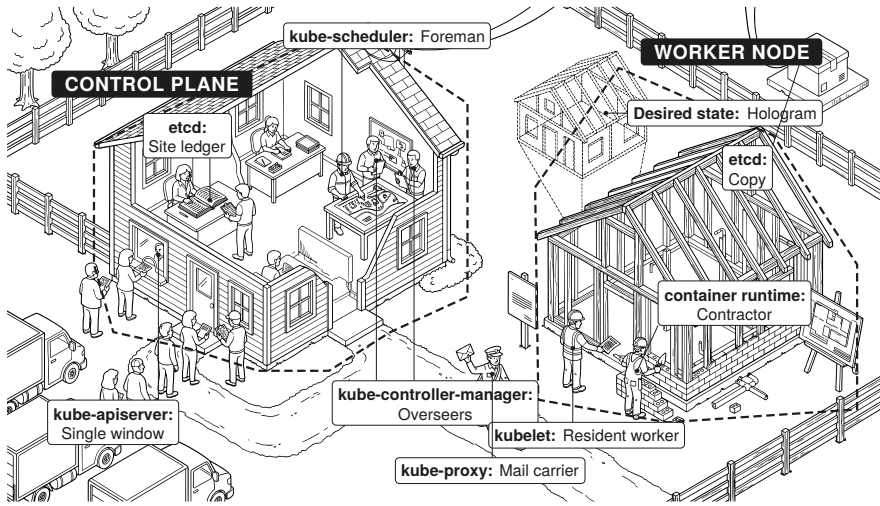


Figure 1.1. The whole estate. On the left the control house: the single window (kube-apiserver), the site ledger (etcd), the foreman (kube-scheduler) and the overseers (kube-controller-manager). On the right a work front: the resident worker (kubelet) reads the order and the contractor (container runtime) is the one who lays the bricks.

This chapter walks the site trade by trade: who commands, who writes things down, who decides, who executes, why the control house is built before anything else, and what happens when the estate grows so much that it needs more than one head office.

Figure 1.1 shows the whole estate at a glance, with every trade at its post: it is worth coming back to it each time a new component steps onto the scene.

1.2 The single window: why everything goes through the kube-apiserver

On a serious building site you don't accept materials from just anybody who shows up with a truck: there is a single window, with a single clerk, who receives every request, checks it, and only then lets it through to the rest of the site. Nobody, not the foreman, not the overseers, not the resident worker in area three, talks directly to the site ledger: everyone talks to the window, and the window is the only one allowed to write in it. Figure 1.2 sums up the whole journey of a request, from your `kubectl apply` until the decision

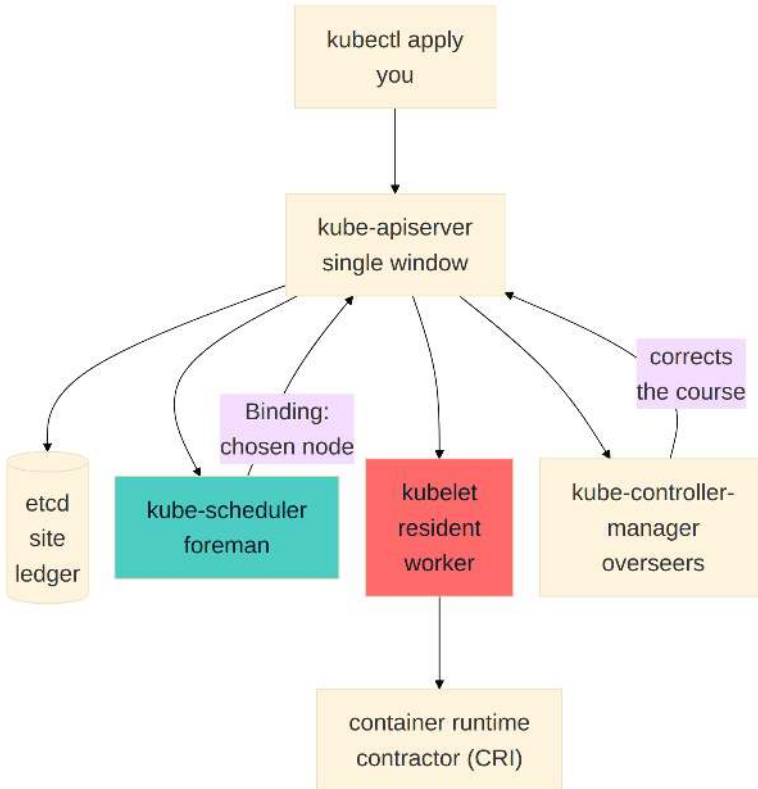


Figure 1.2. The path of a request through the building site: the single window, the record book, the foreman, and the overseers.

is written in the ledger.

In Kubernetes that window is called the kube-apiserver (as Figure 1.3 shows), and the official documentation describes it bluntly as the core component that exposes the Kubernetes HTTP API.

Let's look at the simplest request you can bring to that window. The file `casa.yaml` declares the main house (a namespace) and the people who will live in it (a Deployment with two Pods):

```
apiVersion: v1
```



Figure 1.3. The kube-apiserver, the control house's single window: it receives, checks and lets through every request, and it is the only component that writes to etcd.

```yaml
kind: Namespace
metadata:
  name: casa-principal
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: habitantes
  namespace: casa-principal
spec:
  replicas: 2
  selector:
    matchLabels:
      app: habitantes
  template:
    metadata:
      labels:
        app: habitantes
    spec:
      containers:
      - name: habitante
        image: nginx:1.29
```

```
kubectl apply -f casa.yaml
```

```
namespace/casa-principal created
deployment.apps/habitantes created
```

When you run `kubectl apply -f casa.yaml`, that instruction goes straight to the kube-apiserver, and only the kube-apiserver decides whether to let it through and write it into etcd. You don't talk to the scheduler or to any node: you hand the plan in at the window, and the rest of the site learns about it from there.

Nor does the kube-scheduler place a Pod by chatting with the kubelet of the chosen node: it reports its decision to the kube-apiserver, which writes it down. The kubelet, on the other side, gets no orders under the table: it watches the kube-apiserver, asking what is due in its area. Every trade on the site is, deep down, a client of that same window, and none of them is allowed to skip it.

1.3 The brains of the site: the ledger, the foreman, and the overseers

Behind the single window work three trades that never get mixed, even though they share the same head office.

The first is the site ledger, and in Kubernetes it is called etcd (Figure 1.4 shows it at its desk). The official documentation describes it as a consistent and highly available key-value store for all of the kube-apiserver's data, and that sentence deserves to be read word for word: etcd does not keep application logs, or records of what your containers did, or anything like a complete site diary. It keeps the desired state (the dashed-line house floating over the site in Figure 1.7) and the known state of the cluster, not one line more, and it keeps them consistently: if two overseers ask at the same moment how many workers are on the job, both get the same answer, never two different versions of the truth.

The second trade is the foreman, the kube-scheduler (Figure 1.5 has him



Figure 1.4. etcd, the site ledger: the record keeper writes the desired state and the known state of the estate into the book, always consistently.



Figure 1.5. The kube-scheduler, the foreman: he points on the estate map to the node that gets each Pod that has nowhere to go yet, and records his decision through the window.

pointing at the estate map), and his job is narrower than many people think: the documentation says it watches for Pods that are not yet assigned to any node and assigns each one a suitable node. That is all. The foreman builds nothing with his own hands, does not talk to the resident worker, and doesn't even stay to watch the job get finished: he decides, records his decision in the ledger through the window, and moves on to the next Pod in line.

This is worth a pause. It is very common, especially among people preparing for the exam, to imagine that the kube-scheduler radios the kubelet on the winning node and says "start this Pod now". That never happens, and believing it sends you looking for the fault in the wrong place when a Pod won't start. The scheduler never communicates with a kubelet: it writes a Binding object to the kube-apiserver, which in turn is reflected in etcd, and it is the kubelet on that node that, watching the window, discovers it has been assigned new work and gets it going. The foreman decides; the resident worker



Figure 1.6. The kube-controller-manager, the overseers: clipboards in hand, they compare what the ledger recorded with what is on the ground and correct the difference.

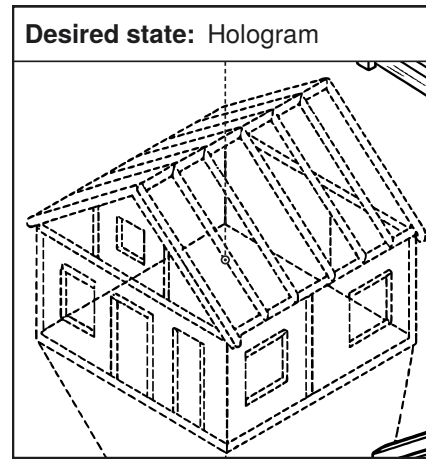


Figure 1.7. The desired state: the hologram of the finished house floats over the site and is the goal everyone compares against; in Kubernetes it is what the manifest declares and etcd stores.

finds out on his own, without anyone calling him in person.

The third trade is the team of overseers, the kube-controller-manager (Figure 1.6 shows them studying that same map), and the keyword there is continuous watching. The official documentation says it runs the controllers that implement the behaviour of the Kubernetes API: one overseer counts how many residents each house has and asks for more if some are missing, another checks that every trade has its service account, another watches the nodes and raises the alarm if one stops responding. None of them builds anything either: all of them compare what the ledger recorded against what stands on the ground, and when the two don't match, they correct the course by writing a new instruction, once again through the single window.

> **Note**
> A Pod stuck in `Pending` with `<none>` in the NODE column of `kubectl get pod <name> -o wide` never received its Binding: the problem sits on the scheduler's side (not enough resources, an impossible nodeSelector, a taint), and `kubectl describe pod <name>` confirms it with a `FailedScheduling` event. If the NODE column already shows a node and the Pod still won't start, the foreman did his part and it is time to look at that area's kubelet and container runtime.

1.4 The trades in each area of the site: the resident worker, the mail carrier, and the contractor

Let's go down from the head office to each area of the site, that is, to each node. Three more trades work there, and these ones do have their hands in the mud every day.

The first is the kubelet, which the documentation describes as the agent that makes sure Pods are running, including their containers. It is the resident worker of that particular area: it watches the kube-apiserver asking which Pods were assigned to its node, and when it finds new work, it carries it out. But the kubelet doesn't lay brick with its own hands either: to create and run a container, it hands the order to a trusted contractor, the container runtime (containerd, for example; Figure 1.8 shows the worker reading the order next to the contractor laying the wall), through a standard contract called CRI, the Container Runtime Interface. That contract is what lets Kubernetes switch contractors without rewriting the resident worker: any runtime that speaks CRI will do.

So let's be clear about who lays each brick. The resident worker takes the order, sends the contractor, and afterwards checks that what was built is still standing. The contractor is the one holding the trowel and the plumb line: it pulls the container image, raises its isolation walls, hooks up its resource lines and starts it. If a container falls over, the kubelet notices and sends the contractor back, following the restart policy the Pod carries.

The third trade in each area is the kube-proxy (as Figure 1.9 shows), and the documentation sums it up as the component that maintains network rules on

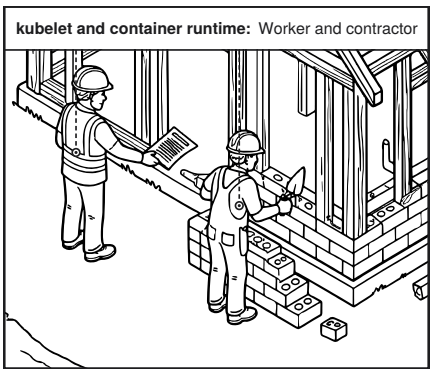


Figure 1.8. The resident worker (kubelet) takes the work order and hands it to the contractor; he never touches a brick. The one who lays them, one by one, is the contractor (container runtime), whom the kubelet hires through the CRI contract.



Figure 1.9. The kube-proxy, the mail carrier of every worker node: it maintains the network rules that take each Service's traffic to one of its replicas.

nodes to implement Services. Think of it as the neighbourhood mail carrier: when someone sends a letter addressed to the Service in area three, it doesn't matter which replica receives it at that moment, the carrier makes sure it reaches one of them, following network rules that he keeps current on every node.

Three trades per area, and one rule that never changes: the kubelet enforces what the head office decided, the container runtime lays every brick following the CRI contract, and the kube-proxy makes sure network mail reaches the right door. None of them decides what gets built; that was settled upstairs, at the single window and in the site ledger.

1.5 Two houses instead of one: the extra cost that prevents ruin

There is one question this whole story leaves hanging: who builds the head office itself? Because the kube-apiserver, etcd, the scheduler and the controller-manager are also processes that have to start somewhere, and they can't depend on a scheduler that doesn't exist yet to place them.

This is where a complexity shows up that you never see when you deploy with Docker Compose. With Compose you raise a single house: one file, one machine, and its containers inside. Compose can restart a container that falls over, with a restart policy, and it can flag it as sick with a healthcheck, but all of that happens on that same machine. There is no head office keeping track of the desired state of the whole estate, no foreman spreading work across several machines, and if the whole machine goes dark, nobody moves its residents to another area of the site. Kubernetes, on the other hand, makes you build two houses: the main house, where your applications live, and the control house, where the window, the ledger, the foreman and the overseers work.

And the order is not up for debate: the control house is built first. Without it, nobody would know who does what on the main house. It is an extra cost, no doubt, and you pay it so the house doesn't come down like the neighbour's, taking with it an investment that took years of work to put together.

A comparison helps show what that extra cost buys. In 2023, Seymour, Masic and colleagues published a study of ancient Roman concrete in *Science Advances*, "Hot mixing: Mechanistic insights into the durability of ancient Roman concrete". They analysed samples from the archaeological site of Privernum and proposed that the small lime lumps in that mix act as reservoirs of reactive calcium, able to fill pores and cracks over time. They then made modern mixes containing those same lumps and demonstrated their self-healing potential. A modern farmhouse doesn't do that: if a wall cracks, you have to call in a builder to find the damage and fix it. Kubernetes is closer to Roman concrete. Watch what happens if someone removes one of the residents of the main house (your Pod names will differ):

```
kubectl -n casa-principal delete pod
 ↪  habitantes-86564cc6b9-4jzmk
kubectl -n casa-principal get pods
```

```
pod "habitantes-86564cc6b9-4jzmk" deleted from
 ↪  casa-principal namespace
NAME                           READY   STATUS
 ↪  RESTARTS   AGE
habitantes-86564cc6b9-mrxkf    1/1     Running   0
 ↪  3s
habitantes-86564cc6b9-vnw2h    1/1     Running   0
 ↪  149m
```

Nobody called the builder: the Deployment's overseer saw two residents on the plan and only one in the house, and replaced the missing one.

> **Caution**
> Self-healing has an owner. A Pod created by hand with `kubectl run visitante --image=nginx:1.29` belongs to no controller: if it is deleted, or its node disappears, nobody replaces it. On a kind cluster, `kubectl delete pod visitante` left it gone for good, while the Pods of the Deployment next to it came back on their own. In production, anything that must survive is declared with a Deployment, a StatefulSet or a DaemonSet, never as a bare Pod.

1.6 A site with more than one head office: static Pods and high availability

Back to the open question: if the control house comes first, who raises it? kubeadm solves that chicken-and-egg problem with static Pods. A static Pod, according to the official Kubernetes documentation, is a Pod that the kubelet of one specific node manages directly, reading manifests from a fixed folder on the file system (normally `/etc/kubernetes/manifests/`), without the kube-apiserver or the scheduler taking part in the decision. It is the only job the resident worker commissions on his own, without waiting for

instructions from the head office, precisely because what he is raising is that office. This is what the folder looks like on the control node of a cluster built with kubeadm:

```
ls /etc/kubernetes/manifests
```

```
etcd.yaml
kube-apiserver.yaml
kube-controller-manager.yaml
kube-scheduler.yaml
```

Four files, four trades of the control house: the kubelet reads them at startup and keeps them alive for as long as they stay on disk.

What you see with `kubectl get pods -n kube-system` is only a reflection of those files, a mirror Pod the kubelet publishes so the window knows what is running. If you delete it with `kubectl delete`, the kubelet brings it right back, under the same name, because the manifest is still in its folder. A Deployment's Pod comes back too, as we saw in the previous section, but by another road: one of the controller-manager's overseers replaces it, through the window and under a new name. The static Pod, by contrast, is replaced by the local kubelet without asking anyone.

And when the estate grows and a single head office is no longer enough, you have to decide how to spread the site ledger across several. The official kubeadm documentation on high-availability topologies describes two ways to do it. In the stacked topology (stacked etcd), each control plane node keeps its own etcd member in the same office where the kube-apiserver lives: it is simpler to set up, but if that office goes down, the control plane and that node's copy of the ledger are lost together. In the external topology (external etcd), the ledger moves to a separate warehouse, on nodes dedicated only to etcd: it costs more machines, but a fire in the head office no longer takes the ledger with it. Neither is "the right one": stacked is what kubeadm sets up by default, and external is what you choose when losing both things at once is not an option.

> **Tip**
> To tell in seconds whether a Pod is static, ask who owns it: `kubectl -n kube-system get pod kube-scheduler-controlplane -o jsonpath='{.metadata.ownerReferences[0].kind}'` answers `Node`, while a Deployment's Pod answers `ReplicaSet`. The name gives it away too, because it ends with the node's name (here, `controlplane`). To change or retire a static Pod you edit or move its file in `/etc/kubernetes/manifests` on that node: `kubectl delete` only removes the reflection, and the kubelet publishes it again.

1.7 Practice in the lab

Topics from lab `lab-part-01` (Part I) for this chapter:

- List the Pods in the `kube-system` namespace and recognise the four trades of the control house.
- Get into the control node and count the manifests in the static Pod folder.
- Find out who is listed as the owner of the kube-apiserver Pod and where its configuration came from.
- Compare that result with a kube-proxy Pod, which a cluster controller (a DaemonSet) does hand out.

1.8 Summary

Back to the estate. The foundations of the house are not laid by a single trade. There is a single window, the kube-apiserver, through which every instruction passes before it gets written down, starting with your `kubectl apply -f casa.yaml`. There is a site ledger, etcd, which keeps the state of the cluster and nothing more. There is a foreman, the kube-scheduler, who decides which node receives each new Pod without ever speaking in person to whoever will build it. And there are overseers, the kube-controller-manager, who never stop comparing what the ledger says against what stands on the ground, and step in when the two don't match.

In each area of the site, the kubelet takes what it was assigned and sends

the contractor, the container runtime, which lays every brick through the CRI contract; the kube-proxy, the neighbourhood mail carrier, delivers network traffic to the right door. Unlike Docker Compose, which raises a single house on a single machine, Kubernetes asks you for two: the main house and the control house. The control house is built first, with static Pods that the kubelet of each control plane node starts without depending on the scheduler, and that extra cost buys something close to Roman concrete: a building that seals its own cracks without anyone having to call the builder.

When the site grows, you have to decide where the ledger lives: stacked alongside the control plane, simpler but more fragile, or in a separate external warehouse, more expensive but safer. With the house standing, the next chapter brings out the keyring: who gets a copy of which key to this same house, and why nobody should carry the master.

Bibliography

- Kubernetes documentation, Kubernetes Components https://kubernetes.io/docs/concepts/overview/components/
- Kubernetes documentation, Static Pods https://kubernetes.io/docs/tasks/configure-pod-container/static-pod/
- Kubernetes documentation, Options for Highly Available Topology https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/ha-topology/
- Seymour, L. M., Maragh, J., Sabatini, P., Di Tommaso, M., Weaver, J. C., Masic, A. (2023). Hot mixing: Mechanistic insights into the durability of ancient Roman concrete. Science Advances 9(1), eadd1602 https://doi.org/10.1126/sciadv.add1602
- Docker documentation, Compose file reference: services, restart https://docs.docker.com/reference/compose-file/services/#restart