Here's What's Actually Going On with Kubernetes Application Development

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



Luis Felipe Ariza Vesga

Here's What's Actually Going On with Kubernetes Application Development

Here's What's Actually Going On with Kubernetes Application Development

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

Luis Felipe Ariza Vesga

Here's What's Actually Going On with Kubernetes Application Development
The complete Kubernetes application development 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: 9798161944721

For the Eskimo curlew (Numenius borealis), which lived on the Arctic tundra of western Canada and Alaska in populations of millions, and which the IUCN now classifies as critically endangered and possibly extinct. And for anyone who deploys an application on a cluster they do not administer: having many replicas does not protect by itself; what protects is the rest of the system absorbing the loss of one.

Contents

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

Preface **5**
What This Book Is About 7
What This Book Is Not 7
How This Book Is Organized 8
About the Commands and Code Examples 8
How to Contact Us 9
Acknowledgments 10

I Application Design and Build (20%) **13**

1 The Recipe You Can Take to Any Kitchen **15**
1.1 The Recipe That Never Got Written Down 15
1.2 The Written Recipe: What a Container Image Actually Is 17
1.3 Cooking the Recipe Once: Layers, and Why the Work Never Repeats . 19
1.4 The Name on the Storefront: Tags, Digests and the Registry Where It Lives 23
1.5 Practice in the lab 32
1.6 Summary: The Recipe That Arrives Intact at Any Kitchen 32
Bibliography 33

Conventions Used in This Book

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

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

This book addresses you respectfully, the way you speak to someone you work with side by side.

Every chapter opens with an everyday scene. That scene is not decoration or padding: it **is** the technical concept told another way, and it comes 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.

This book translates technical language into everyday language, with situations from daily routine that everyone recognizes, used as analogies to understand quickly a Kubernetes concept that is usually explained in purely technical terms. That includes facts such as a failed readiness probe taking the Pod out of the Service's traffic without restarting the container. The goal is that you understand. That is why the analogy never replaces rigour: it is always followed by the official definition, quoted verbatim, 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 `Deployment` and `CronJob`, YAML fields such as `restartPolicy` and `revisionHistoryLimit`, commands such as `kubectl rollout history`, and paths such as `/var/log/pods/`.

Bold marks the first appearance of an important term, and also the words the exam tends to use verbatim. When you see a bold word inside a definition, it is because that exact word matters.

Angle brackets mark what you must replace: in `kubectl rollout status deployment/<name>`, you type the real name of the Deployment 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: a command to verify, 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 CKAD has none: it is a hands-on exam that assigns tasks and grades the final state of the cluster. 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, almost always with a way to check it yourself. It can be skipped without losing the thread. Example: `kubectl rollout history deployment/<name>` shows the revisions a Deployment keeps, and `--revision=<number>` shows the Pod template of one specific revision.

 **Tip**

A shortcut for the exam terminal, a mnemonic or a faster way of doing something. Example: `kubectl create deployment kitchen --image=nginx --replicas=2 --dry-run=client -o yaml` writes the manifest skeleton 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 done. If you are only going to read one kind of box before sitting the exam, make it these. Example: a Job does not accept `restartPolicy: Always` on its Pod template; Kubernetes only allows `Never` or `OnFailure`. If you need something that runs forever, that is a Deployment.

 **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 restrictive egress policy also governs your application's DNS lookup. If it only allows traffic toward the database Pod and forgets egress toward the cluster DNS, the application cannot even resolve the name of the Service it wants to talk to.

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/. It also cites the Helm documentation (https://helm.sh/docs/), which the Helm site states is distributed under CC-BY-4.0, and it links to the Docker and Kustomize documentation, always with a link to each page.

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 or by the other projects cited.

Kubernetes and Helm are among the registered trademarks of The Linux Foundation, and so is the name of the Certified Kubernetes Application Developer certification, according to its official trademark list. This book is independent study material, not affiliated with, endorsed by, or sponsored by The Linux Foundation, the Cloud Native Computing Foundation or any of the projects or companies mentioned.

Preface

How this book is organized, the voice it explains the CKAD exam in, and why you rent the storefront instead of running the whole farm.

This book comes from the same question that started the whole "Here's What's Actually Going On" series: what if the exam of the moment were explained with the same "here's what's actually going on" voice used to explain life in any corner business? The Certified Kubernetes Application Developer exam gets painted as a syntax exam: YAML manifests, probes, ConfigMaps, Secrets, NetworkPolicies, words that sound like a programming course and nothing else. But here's what's actually going on: underneath that vocabulary, building applications for a Kubernetes cluster looks a lot like opening a food business in a rented storefront inside a shopping center someone else runs: you did not put up the building or hire the guard at the front gate, but you still have to pick the recipe that works in any kitchen, decide what kind of business you are opening, change shifts without closing the doors, and answer a delivery that actually knows how to arrive. That is the thread running through this book's fifteen chapters: you build and deploy applications, you do not administer the whole farm, and that distinction changes everything you need to know for the exam.

This is the third book in the series. If you are studying for the exam and want a logical order, start with book 1, "Here's What's Actually Going On with Kubernetes", which explains the foundations of Kubernetes. Continue with book 2, "Here's What's Actually Going On with Kubernetes Administration", which covers what matters most about installing a cluster from scratch and the challenges an infrastructure engineer faces to make the application lifecycle easier for developers. This book 3 is grateful for all that knowledge and focuses on what matters to the developer: the code and control over the application lifecycle in Kubernetes. Here you think like a developer with an SRE and DevOps mindset. You do not worry about the cluster working, because

book 2 took care of that. You worry about the application being deployable, scalable, observable and continuously delivered. When a topic was already explained in the earlier books, such as RBAC or NetworkPolicies, it is recalled here in a few lines and seen from the application's side.

The official CKAD curriculum, published by the CNCF, describes what that developer can do in the cluster:

- Define, build and modify container images.
- Choose the right workload resource: Deployment, DaemonSet, CronJob and others.
- Design multi-container Pods, with patterns such as sidecar and init.
- Use persistent and ephemeral volumes.
- Implement deployment strategies such as blue/green or canary with Kubernetes primitives.
- Perform rolling updates with Deployments.
- Deploy existing packages with Helm and adapt manifests with Kustomize.
- Handle API deprecations.
- Implement probes and health checks.
- Monitor applications with the command line tools, read logs and debug.
- Extend Kubernetes with CRDs and operators.
- Understand authentication, authorization and admission control.
- Declare requests, limits and quotas.
- Use ConfigMaps, create and consume Secrets, and understand Service Accounts.
- Apply application security with SecurityContexts and capabilities.
- Understand NetworkPolicies, provide access through Services and expose applications with Ingress.

The curriculum does not include CI/CD tools as such. It does include what those pipelines use to deploy: declarative manifests, Helm, Kustomize and controlled rollouts. This book treats them as the building blocks of that continuous delivery.

What This Book Is About

This is a preparation book for the CKAD certification exam, Certified Kubernetes Application Developer, from the Cloud Native Computing Foundation (CNCF) and the Linux Foundation. It follows the official v1.37 curriculum, organized into five parts that follow the exam's five current domains: Application Design and Build; Application Deployment; Application Observability and Maintenance; Application Environment, Configuration and Security, the heaviest domain; and Services and Networking. There are fifteen chapters, each opening with an everyday scene about a business in a rented storefront (the recipe that works in any kitchen, the shift change that never closes the shop, the daily menu and the safe, the delivery that actually knows how to arrive) 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 on a real kind cluster with kubectl.

A note about the version. At the time of this edition, the curriculum published by the CNCF is v1.37, with the same content as v1.35, and the official certification page states that the exam is based on Kubernetes v1.37. The same page explains that the exam environment is aligned with the most recent Kubernetes minor version within approximately four to eight weeks. Before booking your exam, check that page and the github.com/cncf/curriculum repository: if the curriculum changed, that page takes precedence over 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; the official documentation at kubernetes.io/docs exists for that, and this book cites it in every chapter. It is also not a containers-from-zero course: it assumes you already know what a container image is, have already built at least one Dockerfile, and already know your way around a terminal and how to read a YAML file without that being explained here. It is also not a manual for a specific Kubernetes distribution or a particular cloud provider: the book stays inside what the CKAD exam actually evaluates on a generic cluster, not how to use one provider's container registry or managed service. And it definitely does not reproduce real exam questions or questions from commercial simulators like killer.sh: that would violate

both the exam's ethics and its confidentiality agreement, and it would also be bad business, because memorizing a question does not help you the day the exam asks the same thing with a different namespace name.

How This Book Is Organized

The book follows the order of the exam's five official domains, grouped into five parts with their declared weight. Part one covers application design and build: the recipe that works in any kitchen, about portable container images; choosing what kind of business to open, about Deployment, StatefulSet, DaemonSet and Job; and the kitchen helper, about init containers and shared volumes. Part two covers deployment: the shift change that never closes the shop, about controlled rollouts; testing the new dish on half the customers, about progressive deployment strategies; and the franchise with assembly instructions, about Helm and Kustomize. Part three covers observability and maintenance: the open sign and the business's pulse, about liveness and readiness probes, and the complaint book and the catalog that changed pages, about logs and the API deprecation policy. Part four, the heaviest domain on the exam, covers environment, configuration and security: how much space each stall in the market asks for, about requests and limits; the daily menu and the safe, about ConfigMaps and Secrets; asking for a service the neighborhood never had, about custom resources; and the employee badge, about ServiceAccounts and security contexts. Part five, the last one, covers services and networking: the delivery that actually knows how to arrive, about Services; the shopping center's main entrance, about Ingress; and who you are actually allowed to talk to from the storefront, about NetworkPolicies. Within each part, chapters move from the most fundamental to the most specific. The glossary appendix gathers the book's key terms for a final review before the exam, and each part's lab assumes you already read that part's chapters, not the chapters of parts still ahead.

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 with resource names taken from the cluster's own OpenAPI schema, not invented pseudocode dressed up to look nice. Whenever an example uses kubectl, the book

tells you the precise verb (create, apply, exec, logs, rollout) and the resource it acts on, because confusing an imperative `kubectl create` with a declarative `kubectl apply` leaves the cluster in a different state than you expected. When a behaviour depends on the installed version, the book teaches you to check it with `kubectl version` and in the kubernetes.io release notes, instead of hard-coding 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.

All the lab code is published at https://github.com/lfarizav/lo-que-pasa-es-que-desarrollar-en-kubernetes. The repository has two folders: `es/` for the Spanish edition and `en/` for the English edition. Each one holds 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-desarrollar-en-kubernetes.git
cd lo-que-pasa-es-que-desarrollar-en-kubernetes/en
```

Each lab brings up its own isolated kind cluster and writes its kubeconfig to a private file, without touching your everyday kubeconfig or any other clusters you have, so never run it against a production cluster. Each step tells you the objective and the deliverable, and the command sequence is kept apart so you can reveal it after trying on your own. Grading queries the live cluster and does not read your command history. The labs resemble what you will be asked to do, because the CKAD is not multiple choice: 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. Even so, they are not a simulator and they do not reproduce exam questions.

Each chapter closes, right before its summary, with a **Practice in the lab** section that tells you which lab belongs to it and which topics you will work on there; the steps, commands and checks live only in the repository.

How to Contact Us

This book is part of Luis Felipe Ariza Vesga's catalog. If you find an example that does not behave the way the book says, a field that got renamed in a recent Kuber-

netes version, a probe's or a deployment strategy's behavior that changed 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 never gets corrected stops being trustworthy. Kubernetes is a fast-moving project, with new versions every few months and API objects that graduate from beta to stable without much 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 one will never be needed. This book would rather end up corrected than end up pretty, the same way a developer would rather have an honest readinessProbe that warns in time that the business is not ready to serve customers yet, instead of an open sign that keeps lying while the storefront is empty inside.

Acknowledgments

This book invented nothing from scratch, and that is exactly what it wants to be grateful 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 CKAD exam curriculum in the open, and for running the certification this book prepares you for.

To the communities and companies that donated entire projects to the CNCF. Handing a whole project to a foundation, with its code, its name and its governance, instead of keeping it as a private advantage, is what lets a developer anywhere in the world use the same tools without asking anyone for permission.

To the maintainers of Kubernetes, kubectl, Helm, Kustomize, the Gateway API and the container runtimes that appear in these pages. Reviewing other people's code, answering issues at midnight and saying no to a badly thought-out feature is work almost nobody sees and everybody depends on.

To every contributor: to those who write code, but also to those who write and translate the kubernetes.io documentation, 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 pocket 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 thanks, last, to the Eskimo curlew (Numenius borealis), whose story lends the cover its hardest lesson. It was one of the most numerous shorebirds in North America, and intensive hunting in the late nineteenth century drove it into a decline it never recovered from; the IUCN now classifies it as critically endangered and possibly extinct. The lesson for anyone who deploys applications is that having many replicas does not protect by itself: an application survives the loss of one Pod when the rest of the system can absorb that loss.

PART I

Application Design and Build (20%)

1

The Recipe You Can Take to Any Kitchen

Why a container image is a complete recipe any kitchen can follow without asking a single question, and what actually separates a movable tag from an immutable digest.

1.1 The Recipe That Never Got Written Down

Imagine you finally decide to open your own food stall in a rented spot at the plaza de mercado. On day one you want to replicate your mother's exact mute santandereano recipe, the one she has been cooking for twenty years without a single complaint. The problem is she never wrote that recipe down anywhere: it lives in her head, and when you ask her how much cilantro to add, she answers "whatever looks like enough". And your mother, that day, while she stirs the pot, tells you in passing, short and in a serious tone, that in the kitchen order is everything: first whatever takes longest to soften, then the meat, and the potatoes last. You nod without paying it much attention, because your business, that day, is the cilantro. That works fine in her kitchen, with her stove, with the pots she has used forever, but the day you try to cook the same mute santandereano in the new spot, with a different stove and different pots, it comes out different, and nobody quite knows why.

What actually happens is that this exact problem, a recipe that only works in one

specific kitchen because nobody wrote it down completely, is exactly the problem a container image solves. The official Kubernetes documentation defines it this way: a container image "represents binary data that encapsulates an application and all its software dependencies", and images are "executable software bundles that can run standalone and that make very well-defined assumptions about their runtime environment" (official Kubernetes documentation on container images). It is the recipe written down to the last gram: it does not say "as much cilantro as looks right", it says how much, which brand, and at what moment it goes in. Any kitchen that receives that complete recipe produces the same mute santandereano, with no need to call and ask anything.

This chapter follows that recipe from the moment it is written until it lands on the customer's counter: how it gets cooked once and reused without repeating the work (the image's layers), and how it gets a name on the storefront so the customer knows what they are ordering (tags, digests, and the registry where that recipe lives).

And in a recipe, order matters as much as the ingredients. If you add the rice before the water, or the potatoes at the same time as whatever takes hours to soften, the food still comes out, but it sticks to the bottom or ends up with some pieces hard and others mushy. That is what your mother was trying to tell you, and Figure 1.1 sets the two pots side by side. A container image works the same way: the Dockerfile is built instruction by instruction, from top to bottom, and the order you write them in decides how fast the image rebuilds. The official Docker documentation on the build cache sums it up: "If a layer changes, all other layers that come after it are also affected." Writing the instructions in the right order is what following best practices means, and the next section shows you how.

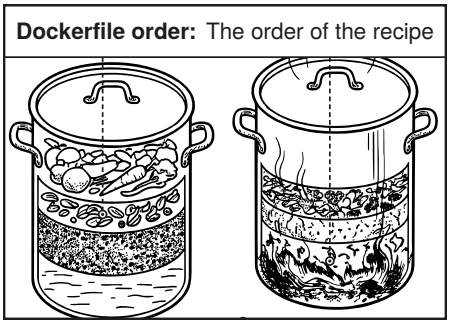


Figure 1.1. Two pots with the same ingredients: in the left one the water goes in first and the rice after; in the right one the order was reversed and the bottom burned. In a Dockerfile, the order of the instructions decides which layers can be reused.

1.2 The Written Recipe: What a Container Image Actually Is

Let's continue with the written recipe. A container image is not a half-finished program that needs someone to fill in instructions at the moment it runs: it is the complete package, ready to run on its own, with no dependency on today's kitchen already having some seasoning or some utensil installed on its own. The same official documentation insists on this point: the image "makes very well-defined assumptions about its runtime environment" (Kubernetes, Container Images), which means the recipe does not assume anything it does not already carry packed inside: if it needs a specific library version, that library goes inside the recipe, not on the kitchen shelf where it might or might not be sitting.

In practice, that recipe is written with a Dockerfile (in Figure 1.2, the booklet where the young cook writes down every step): an ordered list of instructions describing, step by step, how that image gets built from a base image. Each instruction (a `FROM` line, a `RUN`, a `COPY`) adds something concrete to the recipe: the base it is cooked on, a prep step, an ingredient that gets folded in. The comparison to the restaurant is direct: instead of telling the new cook "you already know how a sancocho is made", the written recipe says, in order, chop this much, sauté for this long, add this after that. Nothing is left to whichever cook happens to be on shift, and that is why the same recipe produces the same dish no matter which branch of the chain cooks it.

This is the La Buena Sazón soup-of-the-day recipe written down, with the base image

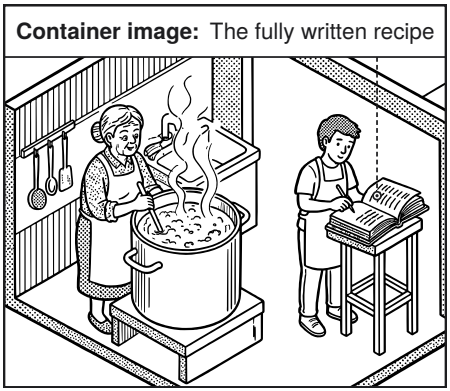


Figure 1.2. The mother cooks from memory while the young cook writes every step into the booklet. That fully written recipe is the container image, and the booklet where it is written down in order is the Dockerfile.

pinned by digest and the `carta.html` file next to it:

```
FROM busybox:1.37.0@sha256:9db7b59979c38555a39def84a31fb9
 ↪  8b5296952f9e3afd4f6f11f05b07adfab0
RUN mkdir -p /carta && echo "La Buena Sazon, sopa del dia"
 ↪  > /carta/fachada.txt
COPY carta.html /carta/index.html
CMD ["httpd", "-f", "-p", "8080", "-h", "/carta"]
```

It is cooked with a single command, which ends with the line `naming to docker.io/library/sopa-del-dia:1.0 done`:

```
docker build -t sopa-del-dia:1.0 .
```

And `docker history` shows the steps that were stored, from the last one to the first:

```
docker history sopa-del-dia:1.0
```

```
IMAGE          CREATED        CREATED BY
 ↪  SIZE      COMMENT
48fa57f12af3   1 second ago   CMD ["httpd" "-f" "-p"
 ↪  "8080" "-h" "/carta"]    0B
 ↪  buildkit.dockerfile.v0
<missing>      1 second ago   COPY carta.html
 ↪  /carta/index.html # buildkit    12.3kB
 ↪  buildkit.dockerfile.v0
<missing>      1 second ago   RUN /bin/sh -c mkdir -p
 ↪  /carta && echo "La B…   20.5kB
 ↪  buildkit.dockerfile.v0
<missing>      2 years ago    BusyBox 1.37.0 (glibc),
 ↪  Debian 13               4.55MB
```

The busybox base arrived ready-made (4.55MB) and on top of it sit two small layers, one for `RUN` and one for `COPY`, plus a `CMD` that takes no space: the recipe records the steps, it does not haul the whole kitchen along.

That fully written recipe is also portable: the same image you build on your laptop runs the same way in the production cluster, on a colleague's machine, or on whichever node the scheduler picked this morning, because none of those destinations has to guess anything the recipe does not already resolve. That portability is why Kubernetes builds its whole deployment unit, the Pod, around containers that run from images, rather than around loose binaries installed by hand on each machine.

1.3 Cooking the Recipe Once: Layers, and Why the Work Never Repeats

A complete recipe is not much use if it has to be cooked from scratch every single time someone orders it. What actually happens is that a container image is not cooked in one single piece either: it is built in layers, and each layer is stored separately, ready to be reused without repeating the work. The official Docker documentation on build best practices explains that, for each Dockerfile instruction, "Docker checks whether it can reuse the instruction from the build cache". If you already cooked the sofrito yesterday and nobody touched that part of the recipe today, there is no need to chop onions again: yesterday's sofrito gets reused as it was stored, like the tray the cook

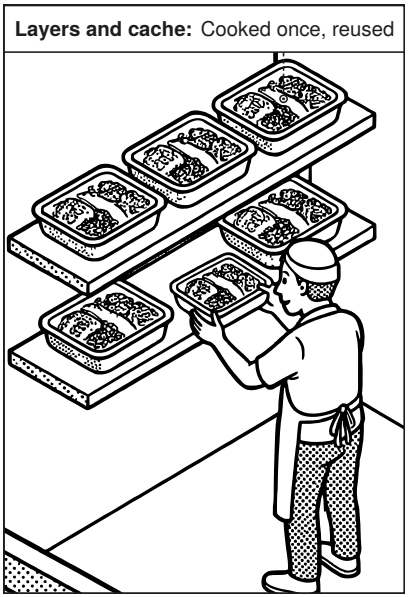


Figure 1.3. A cook takes an already prepared tray from the shelf instead of cooking it again. That is how the build cache works: a layer that did not change is reused exactly as it was stored.



Figure 1.4. Behind the wall stays the prep kitchen, with boards, peelings and pots; only the finished plate comes out through the hatch. That is what a multi-stage build does: the build tooling stays in an earlier stage and the final image carries only what it needs.

takes off the shelf in Figure 1.3.

To watch the cache at work it is enough to change only the menu and build again; the `grep` leaves the three recipe steps in view along with what happened to each one:

```
echo 'sopa del dia de La Buena Sazon, v2 con mas cilantro'
 ↪  > carta.html
docker build -t sopa-del-dia:1.1 . 2>&1 | grep -A1 -E
 ↪  '^#[0-9]+ \[[0-9]/[0-9]\]'
```

```
#5 [1/3] FROM
 ↪  docker.io/library/busybox:1.37.0@sha256:9db7b59979c38
 ↪  555a39def84a31fb98b5296952f9e3afd4f6f11f05b07adfab0
#5 resolve
 ↪  docker.io/library/busybox:1.37.0@sha256:9db7b59979c38
 ↪  555a39def84a31fb98b5296952f9e3afd4f6f11f05b07adfab0
 ↪  0.0s done
--
#6 [2/3] RUN mkdir -p /carta && echo "La Buena Sazon, sopa
 ↪  del dia" > /carta/fachada.txt
#6 CACHED
--
#7 [3/3] COPY carta.html /carta/index.html
#7 DONE 0.0s
```

The `RUN` came out `CACHED` because nobody touched it, and only the `COPY` ran again, because the menu changed. It is yesterday's sofrito taken off the shelf: the only thing chopped today was the new part.

That saving depends entirely on the order in which the instructions are written. The same Docker documentation flags the classic mistake: using `apt-get update` alone, in a `RUN` instruction separate from `apt-get install`, "causes caching issues and subsequent `apt-get install` instructions to fail", because if the cache reuses the `update` line without re-running it, the later `install` can end up installing packages from a stale source list. The official recommendation is blunt: "always combine `RUN apt-get update` with `apt-get install` in the same `RUN` statement". In the food stall: it makes no sense to store "check what is in the pantry" as a separate step from "buy what is missing", a day apart from each other; separate them and you may end up shopping against a pantry list that is no longer today's.

Beyond a single image, when the recipe needs heavy build tooling the final dish does not need (compilers, dev dependencies, full source code), the recommended practice is a multi-stage build. Docker's documentation puts it plainly: multi-stage builds "let you reduce the size of your final image, by creating a cleaner separation between the building of your image and the final output", and they also "let you build more efficiently by executing build steps in parallel". It is the difference between handing

the customer the finished plate and handing them, along with the plate, the whole stack of dirty pots, the cutting boards, and the empty market bags used to prepare it: the prep kitchen stays behind, and only what the customer actually ordered reaches the table, which is what Figure 1.4 shows.

A multi-stage build carries two `FROM` lines in the same file. The `preparacion` stage dirties 20 MB of pots to produce the dish, and the final stage takes only the dish with `COPY --from`:

```
FROM busybox:1.37.0@sha256:9db7b59979c38555a39def84a31fb9
 ↪  8b5296952f9e3afd4f6f11f05b07adfab0 AS preparacion
RUN dd if=/dev/zero of=/ollas-sucias bs=1M count=20
RUN echo "sopa del dia de La Buena Sazon, v3" > /plato.html

FROM busybox:1.37.0@sha256:9db7b59979c38555a39def84a31fb9
 ↪  8b5296952f9e3afd4f6f11f05b07adfab0
COPY --from=preparacion /plato.html /carta/index.html
CMD ["httpd", "-f", "-p", "8080", "-h", "/carta"]
```

Build the preparation stage on its own, with `--target`, and the final image, then compare their sizes:

```
docker build -q -f Dockerfile.etapas --target preparacion
 ↪  -t sopa-del-dia:preparacion .
docker build -q -f Dockerfile.etapas -t sopa-del-dia:2.0 .
docker image ls sopa-del-dia
```

```
sha256:2b16298afbb1c4478a9f2650a32d2d36b89089d03baee1fdaf
 ↪  82cf1cc3139be0
sha256:06cc70a322b29f678756fd95175c5bd25db7ab40dc4b24448d
 ↪  7d579f7fd29ccc
IMAGE                      ID             DISK USAGE
 ↪  CONTENT SIZE   EXTRA
sopa-del-dia:1.0           48fa57f12af3        6.8MB
 ↪  2.22MB
```

```
sopa-del-dia:1.1            0882b9770e3a        6.8MB
 ↪  2.22MB
sopa-del-dia:2.0            06cc70a322b2       6.78MB
 ↪  2.22MB
sopa-del-dia:preparacion    2b16298afbb1       27.8MB
 ↪  2.24MB
```

Then look for the dirty pot in each one:

```
docker run --rm sopa-del-dia:2.0 ls /ollas-sucias
docker run --rm sopa-del-dia:preparacion ls -lh
 ↪  /ollas-sucias
```

```
ls: /ollas-sucias: No such file or directory
-rw-r--r--    1 root     root       20.0M Oct  9 17:01
 ↪  /ollas-sucias
```

The final image takes 6.78MB against 27.8MB for the preparation stage, and the dirty pots did not travel with it: only the dish went out to the table.

1.4 The Name on the Storefront: Tags, Digests and the Registry Where It Lives

With the recipe already cooked and stored in layers, it still needs a name on the storefront so the customer knows what they are ordering. An image is normally identified with a name such as `pause`, `example/mycontainer`, or `kube-apiserver`, and it can also include a registry hostname, for example `fictional.registry.example/imagename`, according to the kubernetes.io page on container images. And if you do not specify a registry, "Kubernetes assumes that you mean the Docker public registry", so a short name always resolves against some registry, even an implicit one: the recipe archive in Figure 1.6.

Now, the name on the storefront is not the only thing that matters, how firm that promise is matters too. The same documentation draws a sharp line between two things that look alike at first glance: "Tags let you identify different versions of the same series of images. Digests are a unique identifier for a specific version of an image.

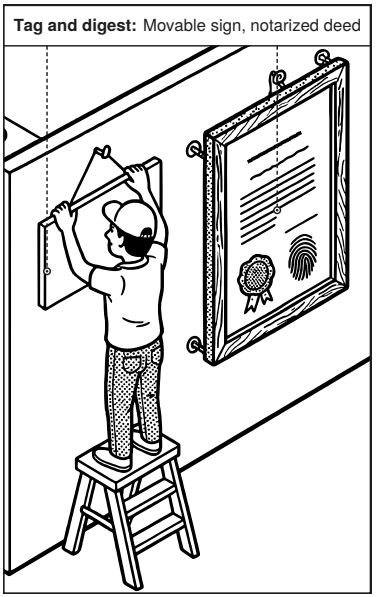


Figure 1.5. A worker lifts the sign off its hooks to hang another one, and next to it the notarized deed, with seal and fingerprint, stays screwed to the wall. A tag can be moved to another image; a digest never changes.



Figure 1.6. At the recipe archive, the clerk hands a sealed box to a delivery rider, in front of shelves of identical boxes. The registry stores the images and hands them to any node that asks for them.

Digests are hashes of the image's content, and are immutable. Tags can be moved to point to different images, but digests are fixed." A tag, then, is like the sign that reads "Restaurant La Buena Sazón" on the storefront: it stays the same sign even if the owner switches meat suppliers next week and the mute santandereano no longer tastes the same. A digest is different: it is that exact spot's notarized deed number, in that exact condition, and there is no way for it to point at anything else afterward; Figure 1.5 shows both, the sign that comes off its hooks and the deed screwed to the wall. That same guarantee is what Docker documents when it explains how to pull an image by pointing directly at its digest, the immutable identifier: pinning the digest on the pull leaves no ambiguity about which bytes are being downloaded, no matter what happened to the tag in the meantime.

The difference fits in four commands: hang the `estable` sign on version 1.0, look,

hang the same sign on 1.1 and look again:

```
docker tag sopa-del-dia:1.0 sopa-del-dia:estable
docker image ls sopa-del-dia:estable
docker tag sopa-del-dia:1.1 sopa-del-dia:estable
docker image ls sopa-del-dia:estable
```

```
IMAGE                  ID             DISK USAGE   CONTENT
 ↪  SIZE   EXTRA
sopa-del-dia:estable   48fa57f12af3        6.8MB
 ↪  2.22MB
IMAGE                  ID             DISK USAGE   CONTENT
 ↪  SIZE   EXTRA
sopa-del-dia:estable   0882b9770e3a        6.8MB
 ↪  2.22MB
```

Same sign, different ID, and nobody told the customer that the recipe behind `estable` changed. The digest, on the other hand, comes from the content itself, and it is what this book uses to pin busybox:

```
docker image ls --digests busybox:1.37.0
```

```
REPOSITORY   TAG       DIGEST
 ↪  IMAGE ID       CREATED       SIZE
busybox      1.37.0    sha256:9db7b59979c38555a39def84a31f
 ↪  b98b5296952f9e3afd4f6f11f05b07adfab0   9db7b59979c3
 ↪  2 years ago   6.77MB
```

That `sha256:9db7b599...` is the notarized deed: whoever pulls by digest gets those bytes, even if tomorrow someone publishes something else under the `1.37.0` tag.

Careful: pinning a specific tag in your manifest, say `my-app:v2`, does not guarantee you get the same bytes every time that image gets pulled. Nothing stops someone from publishing different content under that same `v2` tag later, because tags are movable pointers, not fixed fingerprints. The only real guarantee of "this is the same, byte

for byte" comes from the digest, not the tag; in fact, when `imagePullPolicy` is absent and you do specify the digest, Kubernetes automatically sets the policy to `IfNotPresent`, precisely because a digest is already immutable and there is no need to check it again on every startup.

And speaking of `imagePullPolicy`, that is the last piece: it decides when the kitchen asks the registry for the recipe again instead of using the copy it already has stored. With `IfNotPresent`, "the image is pulled only if it is not already present locally"; with `Always`, every time the kubelet launches a container it "requests the container runtime to pull the image", and the runtime "resolves the image tag or name to a digest, and downloads any layers that are not already cached locally"; with `Never`, "the kubelet does not try fetching the image", and if it is not already present locally the startup simply fails. And when you omit the field entirely, Kubernetes decides for you based on how you named the image: if you used `:latest` or specified no tag at all, the policy is set to `Always`; if you specified any other tag, or a digest, it is set to `IfNotPresent`. That rule explains why two manifests that look similar at a glance can behave very differently the next time the Pod restarts.

In local development with kind, the recipe goes through no registry at all: `kind load docker-image` copies it from your machine's docker into each node's containerd.

```
kind load docker-image sopa-del-dia:1.0 --name
 ↪  ckad-evidence
docker exec ckad-evidence-worker crictl images | grep
 ↪  sopa-del-dia
```

```
Image: "sopa-del-dia:1.0" with ID "sha256:48fa57f12af380a
 ↪  61d9fb86cabe0a068ea7f85a6b9de9f32144c77ada5885435"
 ↪  not yet present on node
 ↪  "ckad-evidence-control-plane", loading...
Image: "sopa-del-dia:1.0" with ID "sha256:48fa57f12af380a
 ↪  61d9fb86cabe0a068ea7f85a6b9de9f32144c77ada5885435"
 ↪  not yet present on node "ckad-evidence-worker",
 ↪  loading...
```

```
Image: "sopa-del-dia:1.0" with ID "sha256:48fa57f12af380a
 ↪  61d9fb86cabe0a068ea7f85a6b9de9f32144c77ada5885435"
 ↪  not yet present on node "ckad-evidence-worker2",
 ↪  loading...
docker.io/library/sopa-del-dia                  1.0
 ↪  0a4a6a5aee0cf       2.22MB
```

The node lists it as `docker.io/library/sopa-del-dia` because containerd normalizes the short name, yet the image was never on Docker Hub. The Deployment that puts it to work in the `la-cocina` namespace declares `imagePullPolicy: IfNotPresent`, so the kubelet uses the copy that is already on the node:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: el-restaurante
  namespace: la-cocina
spec:
  replicas: 2
  selector:
    matchLabels:
      app: el-restaurante
  template:
    metadata:
      labels:
        app: el-restaurante
    spec:
      containers:
      - name: cocinero
        image: sopa-del-dia:1.0
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 8080
```

```
kubectl create namespace la-cocina
kubectl apply -f el-restaurante.yaml
```

```
kubectl -n la-cocina rollout status
 ↪  deployment/el-restaurante
kubectl -n la-cocina get pods -o wide
```

```
namespace/la-cocina created
deployment.apps/el-restaurante created
Waiting for deployment "el-restaurante" rollout to
 ↪  finish: 0 of 2 updated replicas are available...
Waiting for deployment "el-restaurante" rollout to
 ↪  finish: 1 of 2 updated replicas are available...
deployment "el-restaurante" successfully rolled out
NAME                              READY   STATUS
 ↪  RESTARTS   AGE   IP            NODE
 ↪  NOMINATED NODE   READINESS GATES
el-restaurante-6f7c867554-sphpw   1/1     Running   0
 ↪  1s    10.244.2.23   ckad-evidence-worker2   <none>
 ↪  <none>
el-restaurante-6f7c867554-xvnhm   1/1     Running   0
 ↪  1s    10.244.1.14   ckad-evidence-worker    <none>
 ↪  <none>
```

To confirm where the image came from, ask each Pod for its `imageID` and ask the cook for the menu:

```
kubectl -n la-cocina get pods -o jsonpath='{range
 ↪  .items[*]}{.metadata.name}{"
 ↪  "}{.status.containerStatuses[0].imageID}{"\n"}{end}'
kubectl -n la-cocina exec deploy/el-restaurante -- wget
 ↪  -qO- http://localhost:8080
```

```
el-restaurante-6f7c867554-sphpw  docker.io/library/import
 ↪  -2026-10-09@sha256:83a21087dff924682c6a3669729792b65e
 ↪  f26ec6dd797dad0afc797bf9994c9f
```

```
el-restaurante-6f7c867554-xvnhm  docker.io/library/import
 ↪  -2026-10-09@sha256:83a21087dff924682c6a3669729792b65e
 ↪  f26ec6dd797dad0afc797bf9994c9f
sopa del dia de La Buena Sazon, v1
```

The `imageID` says `import-2026-10-09`, which is the fingerprint of `kind load` (an image pulled from a registry would not be named like that), and the cook serves the v1 menu that was packed into the recipe.

Now the counterexample: the same recipe, tagged `latest`, loaded into the nodes the same way, but without writing the policy.

```
docker tag sopa-del-dia:1.0 sopa-del-dia:latest
kind load docker-image sopa-del-dia:latest --name
 ↪  ckad-evidence
kubectl -n la-cocina run plato-del-dia
 ↪  --image=sopa-del-dia:latest
```

```
Image: "sopa-del-dia:latest" with ID "sha256:48fa57f12af3
 ↪  80a61d9fb86cabe0a068ea7f85a6b9de9f32144c77ada5885435"
 ↪  not yet present on node
 ↪  "ckad-evidence-control-plane", loading...
Image: "sopa-del-dia:latest" with ID "sha256:48fa57f12af3
 ↪  80a61d9fb86cabe0a068ea7f85a6b9de9f32144c77ada5885435"
 ↪  not yet present on node "ckad-evidence-worker",
 ↪  loading...
Image: "sopa-del-dia:latest" with ID "sha256:48fa57f12af3
 ↪  80a61d9fb86cabe0a068ea7f85a6b9de9f32144c77ada5885435"
 ↪  not yet present on node "ckad-evidence-worker2",
 ↪  loading...
pod/plato-del-dia created
```

Fifteen seconds later:

```
kubectl -n la-cocina get pod plato-del-dia -o jsonpath='{
 ↪  .spec.containers[0].imagePullPolicy}{"\n"}'
```

```
kubectl -n la-cocina get pod plato-del-dia
kubectl -n la-cocina get events --field-selector
 ↪ involvedObject.name=plato-del-dia -o
 ↪ custom-columns=REASON:.reason,MESSAGE:.message
```

```
Always
NAME            READY   STATUS             RESTARTS   AGE
plato-del-dia   0/1     ImagePullBackOff   0          15s
REASON      MESSAGE
Scheduled   Successfully assigned la-cocina/plato-del-dia
 ↪ to ckad-evidence-worker
Pulling     Pulling image "sopa-del-dia:latest"
Failed      Failed to pull image "sopa-del-dia:latest":
 ↪ failed to pull and unpack image
 ↪ "docker.io/library/sopa-del-dia:latest": failed to
 ↪ resolve reference
 ↪ "docker.io/library/sopa-del-dia:latest": pull access
 ↪ denied, repository does not exist or may require
 ↪ authorization: server message: insufficient_scope:
 ↪ authorization failed
Failed      Error: ErrImagePull
BackOff     Back-off pulling image "sopa-del-dia:latest"
Failed      Error: ImagePullBackOff
```

The policy came out `Always` without anyone writing it, because the tag is `latest`; the kubelet went looking for the image on docker.io even though it had it on the node, the events show the `ErrImagePull` (`pull access denied`), and the Pod was left waiting in `ImagePullBackOff`. Same dish, different sign, and the kitchen does not open.

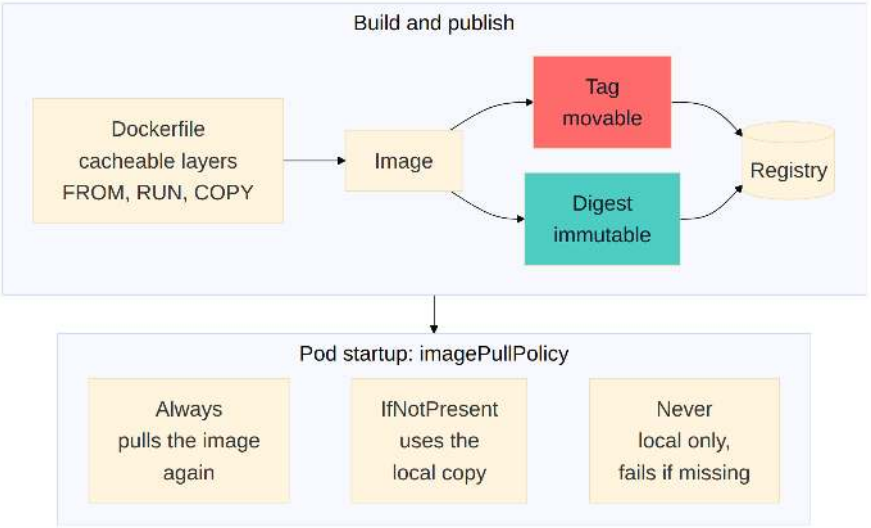


Figure 1.7. The path of a container image: cacheable layers, a movable tag versus an immutable digest, and the three imagePullPolicy behaviors at Pod startup.

> **Warning**
> A tag such as `my-app:v2` does not guarantee you always get the same bytes: it is a pointer someone can republish with different content. Only the digest, a hash of the content, gives that guarantee. To find out which image actually ran, `kubectl get pod <name> -o jsonpath='{.status.containerStatuses[0].imageID}'` returns the reference with its digest, of the form `docker.io/library/nginx@sha256:....`

> **Note**
> The default value of `imagePullPolicy` depends on the tag: it is `Always` if the tag is `:latest`, and `IfNotPresent` otherwise. The field also cannot be changed on a Pod that already exists; to change it, the Pod has to be recreated.

> **Tip**
> When you cannot remember a field, `kubectl explain` is the documentation you have at hand in the exam terminal. `kubectl explain pod.spec.containers.imagePullPolicy` shows the allowed values and the default without leaving the terminal.

1.5 Practice in the lab

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

- Pack the restaurant's recipe: build an image with `docker` from a `Dockerfile` that is already on disk.
- Carry that image into the `kind` cluster with `kind load docker-image`, without borrowing anything from a public registry.
- Put the image to work as a two-replica Deployment, without forcing `imagePullPolicy: Always`.
- Check that the running container comes from that local image and look at the name under which the node stores it.
- Ask the kitchen which menu it is serving and write the result down.

1.6 Summary: The Recipe That Arrives Intact at Any Kitchen

Let's return to the spot in the plaza de mercado. The recipe your mother never wrote down became, in this chapter, a container image: binary data that packages the application and all its dependencies, built to run standalone with no kitchen having to guess anything (Kubernetes documentation, Container Images). That recipe is not cooked in one single piece every time: it is built in layers that Docker reuses from cache whenever the order of the instructions allows it, and when the heavy prep work is useless to the final dish, a multi-stage build leaves that prep kitchen behind and only hands over what the customer actually ordered.

And finally, that recipe needs a name on the storefront: a tag, which is a movable sign that can end up pointing at a different recipe over time, and a digest, which is the notarized deed for that exact version and nothing else. The `imagePullPolicy` setting decides when the kitchen knocks on the registry's door again to ask for the

recipe once more, with default rules that depend on whether you named the image with `:latest`, with no tag, with a fixed tag, or directly with a digest.

With the recipe now clear, the next chapter stops talking about what gets cooked and starts talking about what kind of business gets set up with that recipe: one that lasts a single day, one that never closes, one with reserved tables, or one replicated on every corner of the neighborhood.

Figure 1.7 summarizes that whole path, from the Dockerfile to the imagePullPolicy decision at every startup.

Bibliography

- Kubernetes documentation, Container Images https://kubernetes.io/docs/concepts/containers/images/
- Docker documentation, Building best practices https://docs.docker.com/build/building/best-practices/
- Docker documentation, docker pull (Pull an image by digest) https://docs.docker.com/engine/reference/commandline/pull/#pull-an-image-by-digest-immutable-identifier