An Introductory Textbook for IT Engineers

Cloud Native Textbook

Ver. 1.0.0

open your NEXT future
LPI-JAPAN
Linux Professional Certification
https://lpi.or.jp

Cloud Native Textbook

1.0.0

LPI-Japan Publication

Cloud Native Textbook

Preface

The Non-Profit Organization LPI-Japan is pleased to announce the development and public release of the **Cloud Native Textbook** (hereafter referred to as "this textbook") on the internet, intended for use in cloud native engineer education.

This textbook was developed in response to requests from many educational institutions and engineers for teaching materials and learning environments that allow learners to study cloud native principles and practice from the **basics**.

The textbook is published under the license attached to it (Creative Commons License).

To remain current with the latest technological trends, this textbook will be updated from time to time.

For the most up-to-date information regarding this textbook, please refer to the following webpage:

`https://linuc.org/en/textbooks/cloudnative/`

Purpose of This Textbook

The purpose of this textbook is to help engineers who already meet the **LinuC Level 2** bar understand the structure and principles of cloud native technology and learn through hands-on practice.

Rather than memorizing procedures for Kubernetes or containers, we emphasize grasping **why** each design exists as a structural question.

Areas Not Covered in This Textbook

To keep the practical procedures concise, descriptions of basic Linux operations and system administration (which are covered in the **LinuC Level 1** scope) are kept to a bare minimum. Self-directed research is also part of learning, so some details are intentionally left for you to explore. If something is unclear, please look it up and deepen your understanding on your own.

Intended Learning Environment

This textbook is designed for self-study. Hands-on work is concentrated in Part 2; Part 1 and Part 3 onward are primarily reading material. See Part 2 for detailed procedures.

Part 2 Hands-on (Standard)

Part 2 uses **Killercoda** as the standard hands-on environment. Killercoda is a browser-based environment for learning Kubernetes and related topics. Exercises start from a running Kubernetes cluster and proceed by applying YAML with `kubectl`. You can begin with a Google account or similar, without installing tools or building an environment on your local PC. See Part 2, Chapter 2 for details.

Prerequisites

The network used for exercises is assumed to have **Internet access**, which is required for Killercoda and for referring to this textbook's web page.

Part 2 uses **kubectl**, but the goal is to observe cluster state rather than memorize commands. Basic Linux skills (shell, file operations, and so on) at the **LinuC Level 1** level are assumed.

Local Environment (Optional)

You may also run Kubernetes locally with Minikube, Kind, k3s, Docker Desktop, and similar tools. Behavior and prerequisites vary by environment, so this textbook standardizes on Killercoda. Part 2, Chapter 4 summarizes points to watch when moving exercises to a local environment.

Classroom Use

Even when many learners study together in a classroom, each learner is expected to use Killercoda individually in a browser. If Killercoda is blocked on your network, confirm connectivity in advance. If an instructor provides a shared Kubernetes environment, verify beforehand that Part 2 procedures work there as written.

Overall Flow

This textbook proceeds as follows:

Prologue What Is Cloud Native?

How to use this textbook and why cloud native matters.

Part 1 Fundamentals Step

Containers, Kubernetes, and the difference between cloud and on-premises—understanding the principles.

Part 2 Practice Step

Hands-on work in Killercoda and similar environments to experience how Kubernetes behaves.

Part 3 Application Step

Microservices, continuous delivery, and service-to-service communication—delivering and connecting applications.

Part 4 Operations Step

Monitoring, tracing, and security—making systems visible and keeping them safe.

Part 5 Expansion Step

Business and social context, FinOps, and connections with AI—turning technology into value.

Part 6 Conclusion

A structural recap of what you have learned.

Appendix Bridge to Practice

Certifications such as KCNA and a learning roadmap.

About the Authors and Creators

This textbook is developed as an **open project**. From the planning stage onward, members exchange ideas and share preliminary research, writing, and review.

Toshiyuki Tashibu (SHINESOFT CORPORATION)

We created this textbook hoping it will support everyone who is starting to learn cloud native technology and those who guide that learning. Cloud native spans many technologies and open source projects, so understanding individual tools alone makes it hard to see the whole picture. This book emphasizes structural understanding—why each mechanism was needed and what problem its design solves—rather than a single product or implementation recipe. We hope it serves as a useful entry point into cloud native learning.

Contributors

Tomohiro Katsumura (Nomura Research Institute, Ltd.)

Cloud native technology can feel daunting to start because the domain is broad and the product landscape is large. Yet it is hard to stay competitive in today's business environment without it. We hope this book helps everyone who sets out to learn cloud native technology.

Kanta Koto (Nomura Research Institute, Ltd.)

In daily work I often touched Kubernetes and other cloud native technologies, but I found it difficult to explain systematically what "cloud native" means. Writing this book was a valuable opportunity to reorganize those concepts and connections. We hope it helps readers learn and understand.

Daiki Takasao (Nomura Research Institute, Ltd.)

When I learned Kubernetes and other cloud native technologies, I had no textbook that showed the whole landscape and had to feel my way forward. I joined this project hoping a broad overview textbook would ease that path for others. We hope this book is a good way to start exploring cloud native technology.

Copyright

Rights Regarding Use

Figure 1: CC BY-NC-ND 4.0

Feedback

For feedback on this textbook or inquiries about its use, please contact:

LPI-Japan Secretariat (Specified Non-Profit Organization)

info@lpi.or.jp

Table of Contents

Prologue: What Is Cloud Native?

How to Use This Textbook — For Genuine Understanding

This textbook is not for memorizing operations or procedures.
It is for understanding how to keep a system viable within an environment that continues to change.

1. Why Now

The assumptions surrounding systems have changed.

Change is no longer the exception,
distribution can no longer be avoided,
and uncertainty has become the baseline assumption.

Within this situation, **keeping things viable**
has become harder than building them in the first place.

Cloud native is a structural response to this change.
The reason to learn it is not that it is new technology.
It became necessary because the traditional assumptions can no longer keep systems viable.

2. The Structure of This Textbook

This textbook is organized in the following sequence.

- The prologue establishes the perspective,
- Part 1 builds understanding of the underlying principles,
- Part 2 lets you experience them firsthand,
- Part 3 engages with them as structure,
- Part 4 applies judgment through operations,
- Part 5 connects them to society,
- Part 6 brings it all together as a conclusion.

This sequence is intentional.
Each part is built on the assumptions established in the part before it.

3. What It Means to Understand Through Structure

In this textbook, before tools or procedures,
the focus is on "**why that form exists.**"

Not on memorizing Kubernetes,
but on seeing why it became necessary.

Not on memorizing commands,
but on grasping what is actually happening.

Understanding through structure means looking not at the technology itself,
but at **the responsibility that technology has taken on**.

4. How to Read

It is fine not to understand immediately.

This textbook deliberately avoids explanations that create the illusion of understanding.
Read through it while holding the structural perspective.

Sections that are unclear will look different after moving forward and returning.
What was covered in a previous part will reappear in a different form.

Through that back-and-forth, understanding accumulates.

5. Reproducibility

The structures addressed in this textbook do not depend on any specific product or environment.

AWS, GCP, Azure, on-premise.
Even when the environment changes, the axis of judgment does not change.

What to keep viable.
How far to hold responsibility.
Where to draw the boundary.

These questions do not change.

What Is Cloud Native?

1. The Question

The term “cloud native” does not refer to a collective name for cloud services or container technologies.

Cloud is an environment structure that assumes change, scalability, and automation.
Native means adapting to that environment and continuing to remain viable as a whole.

So, what is cloud native?

2. Background: Not Technological Evolution, but a Change in Assumptions

Monoliths remained viable under fixed assumptions.
Virtualization separated environments, but the assumptions did not change.
Containers made the unit smaller, but the whole was still managed by people.

In cloud native, the assumptions themselves changed.

This is not technological evolution.
It is a change in the assumptions under which systems exist.

Traditional systems assumed that the environment would not change significantly.
Configuration was fixed, and changes and failures were treated as exceptions.

Today, however, services change continuously,
configuration is distributed, and the environment is no longer stable.

With change, distribution, and uncertainty now taken as assumptions,
systems can no longer remain viable under the traditional assumptions.

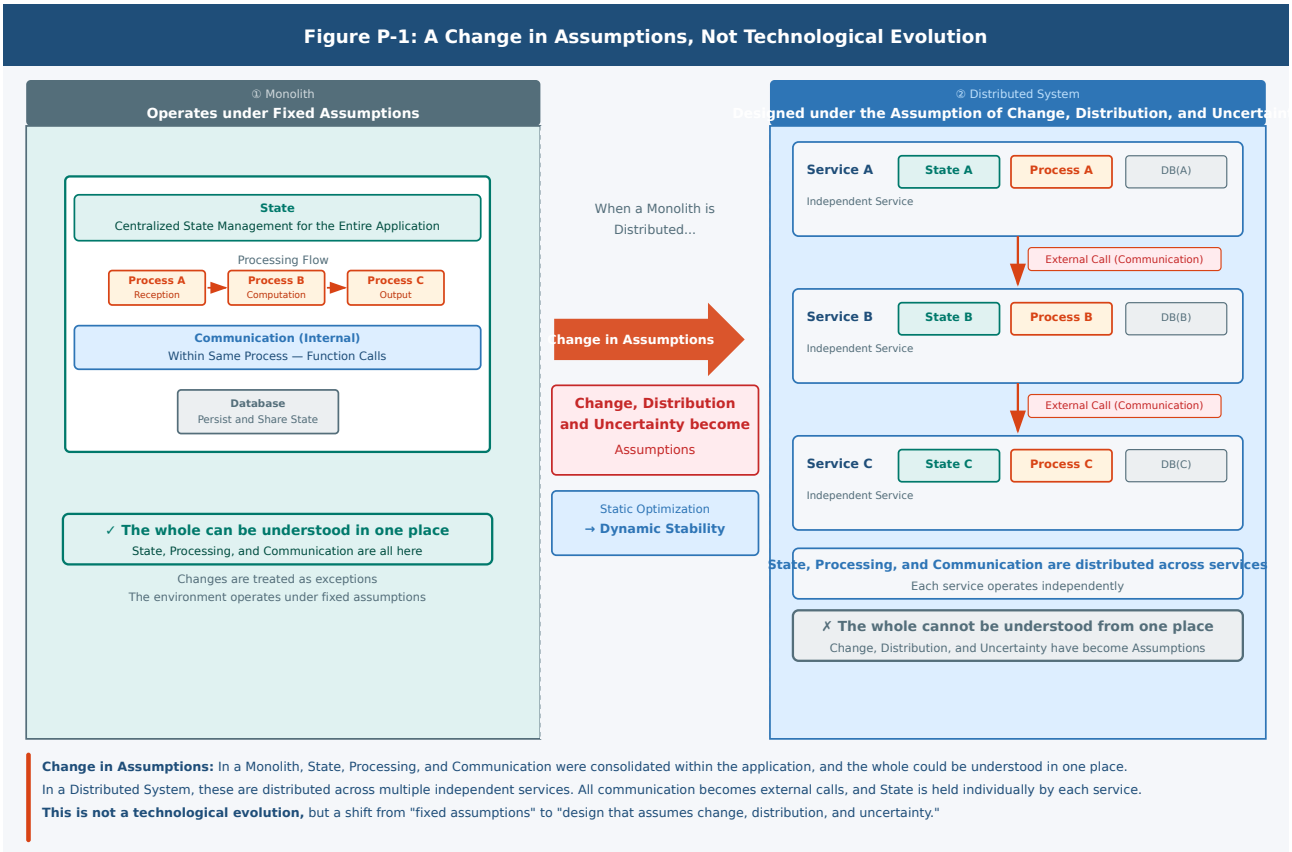


Figure P-1: Not Technological Evolution, but a Change in Assumptions

3. The Core: From Static Optimization to Dynamic Stability

Traditional design was optimized under the assumption that change would not occur.
Configuration was fixed, and failures and changes were treated as exceptions.

In cloud native, the assumptions are inverted.

Change is treated as something that always occurs,
and configuration is designed under the assumption that it continues to move.

Here, one assumption must be made explicit.

Cloud-native design does not assume that things will not break.
It does not assume that everything can be fully controlled.

Parts breaking, state drifting, environment changing —
these are not treated as anomalies, but as assumptions.

Within that, the system is still required to continue to remain viable as a whole.
The goal is not to prevent stoppage, but to ensure that the whole continues to remain viable even when parts stop.

Rather than supporting through operations, absorb through structure.
Rather than avoiding change, treat change as an assumption.

Under this assumption, operations that rely on human judgment case by case cannot keep up.

Therefore, the judgment required to maintain viability
is designed to be carried out by the structure, not by people.

4. The CNCF Definition

The CNCF definition of cloud native is widely referenced.

It cites containers, microservices, declarative APIs, and observability as examples,
and emphasizes characteristics such as loose coupling, automation, scalability, and resilience.

However, this is only one way of organizing the concept.

This textbook treats these not as individual technologies,
but as a structure that assumes change.

These are not individual technologies.

Containers are a means to isolate execution units.
Microservices are a configuration to divide responsibilities.
Declarative APIs are an interface for managing state.
Observability is a mechanism for understanding state and enabling judgment.

All of these are positioned as elements that enable the system as a whole to continue to remain viable,
within an environment that assumes change.

5. What Learning Makes Visible

When cloud native is understood from this perspective,
the way systems are seen changes significantly.

Rather than individual technologies,
what becomes visible is where change occurs and where viability is supported.

The center of design becomes not “making it run,”
but “continuing to remain viable as a whole.”

And judgment comes to be based not on function,
but on structure.

This change is not simply the addition of a skill.
It means that the assumptions about systems themselves change.

For example, when a problem occurs in which a service is slow,
it may not be immediately possible to identify where the cause lies.

Is it a problem in the application?
A problem in the network?
Or a problem in the integration with another service?

Within a distributed structure,
a single cause affects the system as a whole.

What matters in this situation
is not knowledge of individual technologies.

It is understanding where change is occurring,
and where viability is being supported.

This change is not simply the addition of a skill.
It means that the assumptions about systems themselves change.

6. Summary

What has been addressed here
is not an explanation of the term “cloud native.”

It is a perspective on what assumptions to hold about systems.

When standing on this perspective,
the way systems are seen and the approach to design change significantly.

How, then, is that structure realized?
We hope you will proceed to the next part with this question in mind.

Part 1: Fundamentals Step - The Story of How the Execution Unit Changed

Introduction

Cloud-native technologies are already in widespread use.

Containers, Kubernetes, cloud —
many people have already worked with these.

Yet certain situations tend to arise.

- Can run it, but cannot explain why it works that way
- Can replicate configurations, but changes cause things to break
- When problems occur, cannot determine where to look

The issue here is not the technology.
It is the inability to see the whole as a structure.

This part treats cloud native not as a collection of individual technologies,
but as a separated structure.

Four areas are addressed.

- Execution unit (where to draw the boundary)
- Control (how to keep it viable)
- Execution environment (where to run it)
- Assumptions (what it depends on)

Each of these is a distinct problem.

What matters is not understanding everything.
Being able to isolate where something is happening and what is happening there.

Carry this perspective forward into the next chapter.

Chapter 1: Container Basics and Docker - The Story of How the Execution Unit Changed

1. Why Was It Necessary to Reconsider the "Execution Unit" ?

Applications were once treated as things that run on top of a specific environment.

The environment here refers to the collection of assumptions required for an application to run —
such as the OS, dependent libraries, and runtime.

As environments multiplied, changes increased, and the need to run the same things the same way grew, those assumptions began to break down.

The problem was not the application itself.
It was that the assumptions under which it ran were dependent on the environment.

For example, something that ran in the development environment
would not run in the production environment.

- Library versions differed
- Configurations differed

- Dependencies were not aligned

The problem in this situation was not the code.

It was that the assumed environment did not match.
What was called into question here was: at what unit should the application be treated?

2. What Virtualization Changed

Through virtualization, environments were separated from physical servers.

Environments could be duplicated, moved, and restored.
Environments no longer needed to be fixed.

However, the way applications were treated did not change.
The execution unit remained close to the environment.

Environments were isolated,
but applications remained dependent on the environment.

In this state,
“where to run” could be resolved,
but “how to treat” had not changed.

Therefore, it became necessary to change the unit of treatment itself.

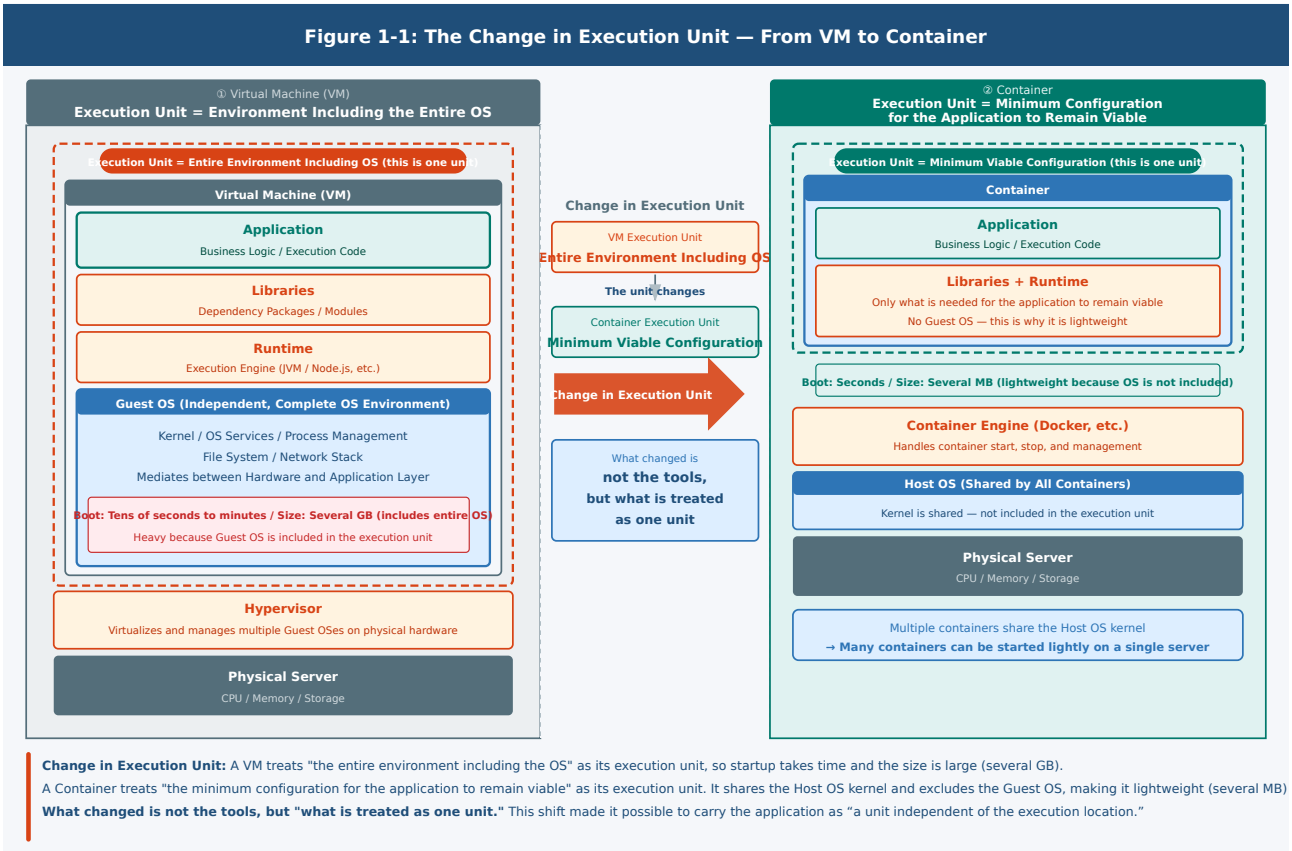


Figure 1-1: The Change in Execution Unit — From VM to Container

3. The Container as an Execution Unit

A container separates the execution unit from the environment.

What is handled is not the entire OS,
but the minimum configuration required for the application to remain viable.

Through this distinction,
an application is treated as a unit that does not depend on where it runs.

For example, if the same container is moved to a different environment as-is,
it can be expected to run in the same way.

Rather than adjusting configurations for each environment,
the idea becomes carrying the viable state as a unit.

Created when needed, discarded when no longer needed.
The assumption becomes replacement, not maintenance of a fixed instance.

4. The Idea of Treating Something as a Unit

What changed with containers was not the tooling.

Applications came to be treated not as “things that run on top of an environment,”
but as **units that contain their own conditions for viability.**

Rather than maintaining the environment,
viable units are continuously replaced.

Through this distinction, the meaning of stability changes.

Stability is not the maintenance of a state.
It is a state of viability achieved through continuous replacement.

5. The Role of Docker

The container mechanism itself is realized through Linux capabilities.
Docker is the representative tool that made this mechanism accessible to everyone.

Through unified operations and a reproducible format, the container concept became widely applicable.

Through unified operations and a reproducible format,
containers became a unit that individuals could handle.

Docker is not cloud native itself.

However, by making it possible to actually work with this unit,
the idea of “treating applications as units” became a reality.

6. Containers and Microservices Are Separate Concerns

Containers address the execution unit.
Microservices address division and responsibility.

How to operate a unit
and how to divide a unit are different problems.

Conflating these two
obscures the intent of the design.

Containers address “how to handle.”
Microservices address “how to divide.”

The problems they address are different.

7. The Question That Remains

The execution unit has been separated from the environment
and can now be treated as a small unit.

As a result, units increase in number, become distributed,
and change and failure become assumptions.

Then, how are these ever-increasing units
kept viable as a whole?

Chapter 2: Kubernetes Overview and Key Components - The Story of Keeping Distributed Execution Units Viable

1. What Became a Problem in a World of More Containers?

With containers, applications could be treated as small units.
As a result, units increased, execution locations became distributed,
and failure became not an exception but an assumption.

In this state, it is not realistic for people to manage each unit individually.
As the number grows, states diverge and understanding the whole becomes difficult.

What becomes a problem in this state is not the behavior of individual applications.
Who, and how, keeps these ever-increasing units viable as a whole?

2. What Does Kubernetes Take Responsibility For?

Kubernetes is a mechanism designed to address this problem.

What it handles is not individual containers.
It is the state in which the whole remains viable.

Rather than people operating individually, maintaining the state in which the whole remains viable becomes the assumption.
Not where something is running, but what state it should continue to be in — that is what is addressed.
So that the state is maintained, scheduling, restarting, and adjustment continue to take place.

3. Managing State and Continuously Closing the Gap

Kubernetes treats the current state and the desired state separately.

Not “what to do,”
but “what should be the case” is defined.

For example, even if one container stops,
if the state “it should be running” is defined,
it will be started elsewhere.

The actual state continues to change at all times.
Therefore, a gap between actual and desired state always arises.

Kubernetes continues to observe that gap
and continues to act to close it.

Rather than aligning once,
it continues to restore under the assumption that drift will always occur.

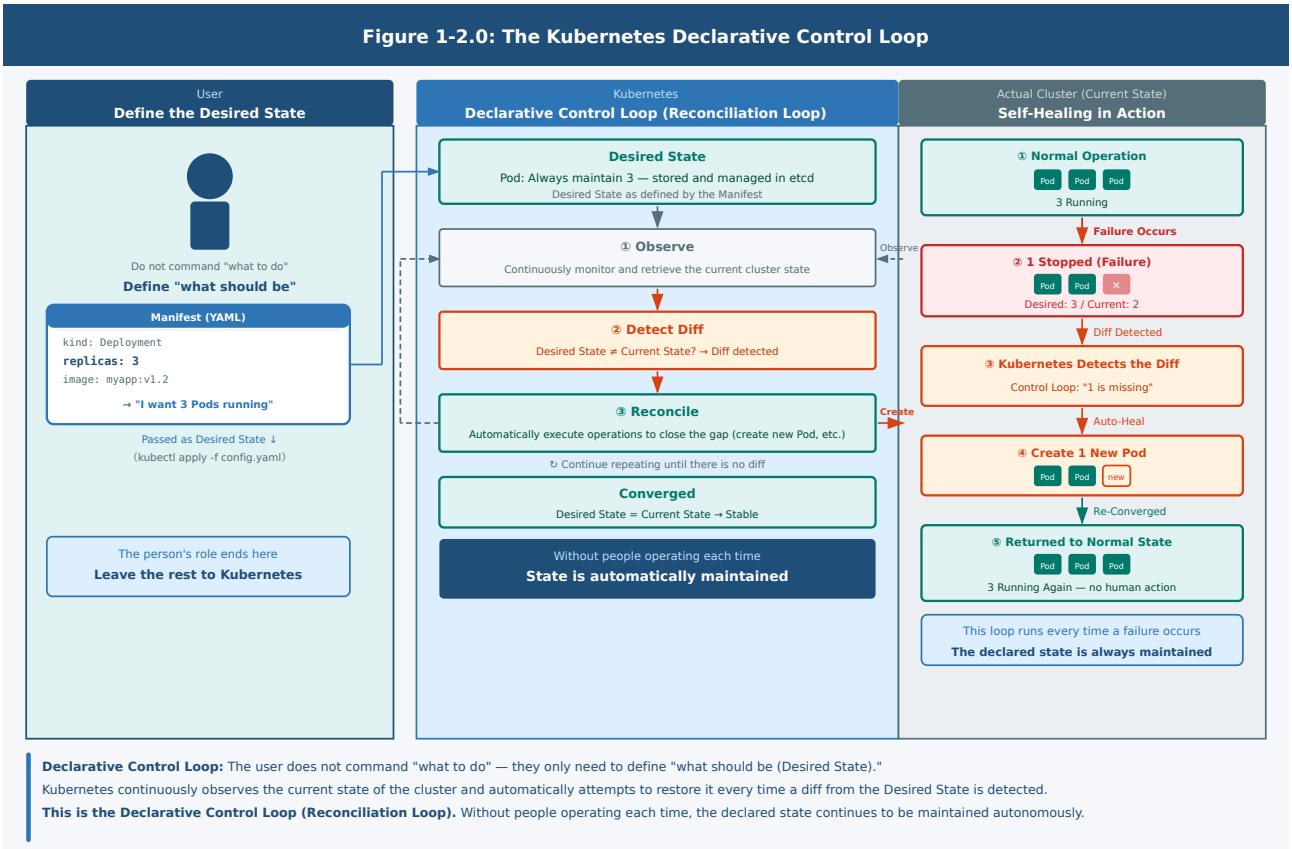


Figure 1-2.0: Kubernetes Declarative Control Loop

4. The Side That Manages State and the Side That Executes

To address this problem, Kubernetes is designed to separate the role of managing state
from the role of actually running applications.

To keep this mechanism viable, the roles are divided broadly into two.

The side that handles state and keeps the whole viable,
and the side that actually runs applications.

The former understands the desired state,
and continues to make judgments so that state is maintained.

The latter receives the resulting instructions
and actually runs containers.

Through this separation,
a structure is maintained in which even if a part changes, the whole does not collapse.

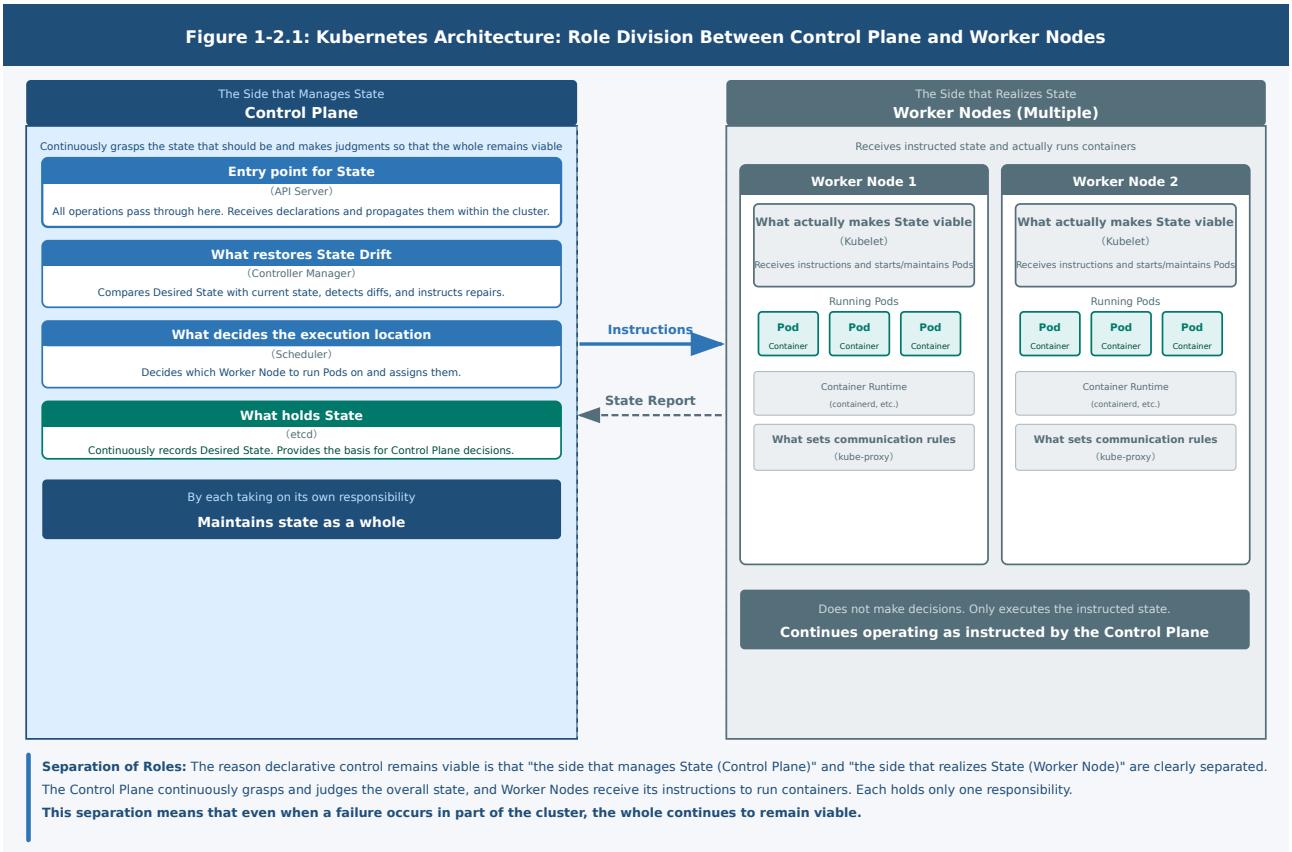


Figure 1-2.1: Role Division Between Control Plane and Worker Node

5. A Structure Divided by Responsibility

This is a design decision in the architecture of Kubernetes.

Within this structure, each role is further divided.

- What serves as the entry point for state
- What decides where to execute
- What detects and corrects drift
- What actually makes the state viable

Each holds only one responsibility
and does nothing beyond that.

Through this separation,
even if a failure occurs in part of Kubernetes, the cluster as a whole continues to remain viable.

6. Kubernetes Is Not an OS

Kubernetes is not an OS.

Because it can handle multiple machines together,
it may appear that way.

In reality, however, it does not directly manage resources —
it is responsible for the control required to maintain the desired state.

Execution itself is delegated to the OS and the mechanisms on top of it.

Kubernetes does not replace those.
It is the layer that keeps the state viable on top of them.

7. The Question That Remains

Up to this point, the mechanism for keeping distributed units
viable as a whole has come into view.

However, one more assumption remains.

Where are those units run?

Is the environment fixed, or does it change?
How much is controlled by the team itself, and how much is left to the mechanism?

This assumption significantly changes the meaning of design.

Chapter 3: Characteristics of Cloud vs. On-Premise - Thinking About Where to Run as a Structure

1. Why Does the Debate Never Converge?

Discussions about cloud and on-premise often reach no conclusion.

This is not because one is superior to the other,
but because the comparison is made without aligning on assumptions.

Even for the same system,
whether change is assumed,
or fixed state is assumed,
changes the basis for evaluation significantly.

When assumptions differ, the appropriate design and judgment differ as well.

Note that the characteristics described here represent general tendencies. It is possible to adopt a fixed configuration on cloud, or to adopt a design that assumes change on-premise.

2. The Assumption of Cloud

Note that the cloud referred to here is in the sense of an execution platform,
and should be understood separately from the design philosophy of cloud native.

Cloud is not a specific service or location.
It is an environment that assumes change, scalability, and automation.

The environment is not fixed.
It scales up and down as needed,
and failure is treated as an assumption.

For example, as load increases, resources increase; when no longer needed, they decrease.

When a failure occurs, rather than repairing individual components,
they are reconstructed elsewhere.

Rather than maintaining the environment,
viability is maintained while accepting change.

3. The Assumption of On-Premise

On-premise is not the opposite of cloud.
It is an environment in which the assumption is that you control it yourself.

Where it runs, in what configuration it runs —
the responsibility for that is taken on by the team itself.

Configuration is intentionally designed,
and changes are treated as managed events.

What to fix and what to change
is left to the team’ s own judgment.

Figure 1-3: The Difference in Assumptions Between Cloud and On-Premise

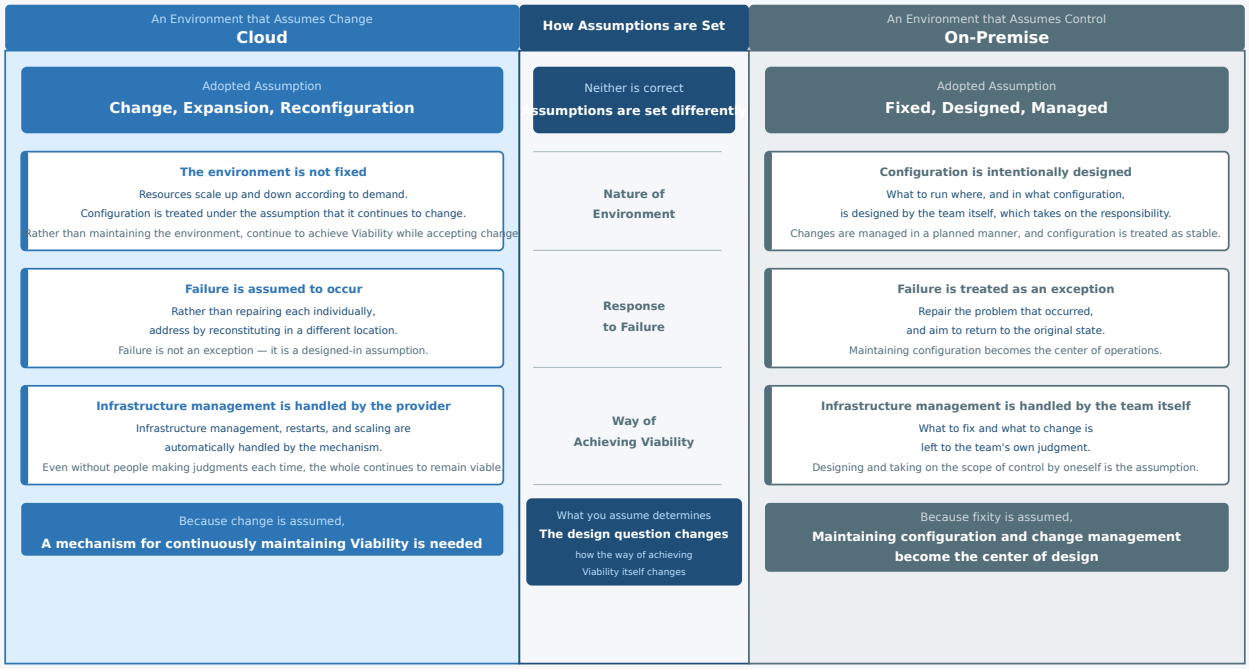


Figure 1-3: The Difference in Assumptions Between Cloud and On-Premise

4. What Is Different?

The difference is not performance or cost.

- What is assumed to change
- What is assumed to stay fixed
- Who takes responsibility

The way assumptions are set differs.

When change is assumed, configuration is treated as something that continues to move.
State changes constantly, and viability must continue to be maintained.

When fixed state is assumed, configuration is treated as something stable.
State is maintained, and changes are treated as managed events.

Neither is correct.

When assumptions change,
how viability is achieved changes as well.

5. Choosing an Assumption

Deciding where to run something
is not a matter of choosing an environment.

It is a matter of deciding which assumptions to adopt.

Whether to accept change,
or to control and fix.

That choice changes the meaning of design.
What is assumed determines what problem is being addressed.

- What is to be kept viable
- How much is left to the mechanism
- How much is taken on by the team itself

The way those questions are framed changes.

6. The Question That Remains

The unit has been separated, and its viability has come to be controlled.
And the environment that serves as its assumption has also been organized.

Then, what supports those assumptions?

Chapter 4: Fundamentals of Linux and Networking - Putting Words to the Foundation That Underlies Everything

1. Why Return to the Foundation Now?

Cloud and Kubernetes appear to be self-contained on their own layer.
In reality, however, everything they do depends on the layer beneath them.

The assumptions have not disappeared.
They have simply become invisible.

Through abstraction, things have become easier to handle,
and with that, opportunities to be aware of those assumptions have decreased.

2. The Assumption of Linux

Much of the mechanism behind both containers and Kubernetes is built on Linux capabilities and concepts.

Processes are isolated,
resources (computational resources such as CPU, memory, and network) are distributed,
and boundaries are maintained.

Through this mechanism,
multiple processes can be handled independently within a single system.

Containers are not a new mechanism.

They use this already-existing assumption
and give it a form that can be treated as an execution unit.

Therefore, they are created when needed
and discarded when no longer needed.

3. Resources Are Not Always Stable

CPU and memory cannot always be used in the same way.

As load changes, behavior changes,
and depending on the situation, processing slows down or stops.

These are not exceptions.

They are unavoidable properties of handling finite resources.
In an abstracted environment, these constraints become difficult to see.

However, they have not disappeared —
they have simply become invisible.

4. The Assumption of Networking

The network does not always connect.

- Latency occurs
- Ordering is disrupted
- Connections drop without warning

For example, when communication is slow,
the cause may not be immediately clear.

These are not exceptions.

They are properties that arise within a finite environment, and are part of its assumptions.

5. Failure Is an Assumption

With containers, units are separated,
and with Kubernetes, they come to be controlled.

However, the foundation on which these run
continues to change and remains uncertain.

A state in which some parts are functioning and others are failing
is not exceptional.

This state cannot be eliminated.

What is needed is
a structure from which recovery is possible when failure occurs.

6. Who Takes Responsibility for the Assumptions?

These assumptions are always taken on somewhere.

Resource constraints and network uncertainty
cannot simply be left unaddressed.

- Does the application take them on directly?
- Does a control mechanism such as Kubernetes absorb them?
- Does the cloud environment abstract them and make them invisible?

Somewhere, they are always taken on.

Where responsibility lies
changes the meaning of design.

7. The Question That Remains

What has been examined up to this point is not new technology.
Everything rests on assumptions that already existed.

Then, in order to treat those assumptions as assumptions,
how is state understood, and how are judgments made?

Part 1 Summary - Understanding the Principles, Seeing It as a Structure

What has been addressed up to this point is not individual technologies.
It is a perspective on how to understand a system.

The first thing addressed was the unit.

Applications came to be treated not as a single mass,
but as separated units.

However, when units are divided, that alone is not enough for viability.
Who, and how, keeps those units viable?

Here, control becomes necessary.
And where does that control run?

Is change assumed,
or is fixed state assumed?

Assumptions change the meaning of design.

And beneath those assumptions lie properties that cannot be changed.

Resources are finite,
the network is uncertain,
and failure is unavoidable.

These have not disappeared.
They have simply become invisible.

Unit, control, assumption, property.

When these four connect,
the structure of the whole finally becomes visible.

What matters is not memorizing all the details of individual technologies.

Being able to isolate and understand
where something is happening and what is happening there.

And there is one more important point.
Those assumptions are always taken on somewhere.

- By the application
- By the control mechanism
- Or by the environment

Where responsibility lies
changes the meaning of design.

Up to this point, everything addressed in Part 1 has come together.

Unit, control, assumption, property.
And the responsibility that takes them on.

When this structure can be understood as an assumption,
individual technologies are no longer seen as scattered knowledge,
but as a single flow.

Then, how does this structure actually operate?

Part 2: Practice Step - Experiencing the Assumptions Firsthand

Introduction

In Part 1, cloud native was understood as a structure.

Units were separated,
viability was achieved through control,
assumptions changed the meaning of design,
and beneath them lie properties that cannot be changed.

However, this understanding
does not fully take hold through words alone.

Even when something is understood as a structure,
actually working with it often reveals that things do not behave as expected.

A Pod was deleted and something started up.
A configuration was changed but it did not take effect.
An error appeared but it was not clear where to look.

The problem in these situations is not an operational error. It is that the perspective for seeing “what is happening”
has not yet been internalized as a felt sense.

Understanding through words
and grasping through behavior are different circuits.

From here, that structure
will be grasped as actual behavior.

The purpose is not to memorize operations.

It is to confirm how the assumptions organized in Part 1
actually appear as behavior.

The approach from here is straightforward.

First, learn what kinds of environments can be used to run Kubernetes.
Then, confirm the assumptions of the environment used in this part.

Next, observe what behavior actually occurs
while working with basic resources.

And finally, with that understanding in hand,
organize what changes when returning to an actual environment.

Carry this perspective forward into the next chapter.

Chapter 1: Options for Practice Environments - Where to Run Kubernetes

1. Why Start with the Environment?

There is not one environment for practicing Kubernetes.

Even with the same operations, results change when the environment is different.
Following the steps exactly but it does not work.
It is written as described but the behavior is different.

In these situations, the problem is not Kubernetes.
The assumptions of the environment being used are simply different.

When the assumptions of the environment differ, what is handled automatically
and what must be handled by the person changes.

Without understanding that difference, “why it does not work” becomes unclear.

2. Aligning Assumptions

What matters is not which environment to use.

It is whether one is consciously aware of what assumptions the environment operates under.
What is automatically provided by the environment, and what must be taken on by the person, changes significantly depending on the environment.

For example, in a local environment,
tools such as Minikube, Kind, k3s, or Docker Desktop are used,
and it is necessary to take on part of the setup and control oneself.

On the other hand, as a pre-configured environment,
there are options such as Killercoda where the execution environment is provided.

Neither is superior.

In this part, in order to focus on “observing behavior,”
the impact of environment differences and assumption gaps is minimized as much as possible.

Therefore, a pre-configured environment is used as the assumption.

What matters is that this choice is not made because it is “convenient,”
but because it allows focus on observation.

■ Main Options for Running Kubernetes

※ Note on the execution environment addressed in this chapter

Minikube, Kind, k3s, and Docker Desktop
are Kubernetes execution environments intended primarily for learning and verification.

On the other hand, public cloud environments (AWS, Azure, Google Cloud)
used in actual service operations
often use environments with different assumptions.

What matters here is not which is superior.
It is that the assumptions differ.

The purpose of this chapter is not to decide “which to use in production.”

The options addressed are those used as learning environments
for running Kubernetes and understanding its behavior and assumptions.

Here, representative environments commonly used when trying out Kubernetes are examined.
What matters is not deciding which is best.

It is understanding the purpose for which each option was created,
and what assumptions it operates under.

3. How This Part Handles the Environment

In this part, the impact of environment differences is minimized.

Before entering practice, it is important to be conscious of the assumptions of the environment.
With those assumptions established, the behavior of Kubernetes is observed.

The purpose is not to memorize operations.
It is to observe the behavior of Kubernetes.

Even with the same operations, why does a particular result occur?
What changes, and what does not?

Capturing that difference is what matters.
It is fine not to understand immediately.

Confirm repeatedly how the assumptions addressed in Part 1 appear.

Chapter 2: Standardizing the Practice Environment - Aligning Assumptions for Observation

1. Why Align the Environment?

In learning Kubernetes, behavior changes depending on the environment.

When the same operations yield different results,
the problem lies in the assumptions of the environment, not in Kubernetes.

Proceeding without addressing this leads to not knowing what is being observed.

Much of the time spent on things unrelated to the essence of learning
comes from this gap in assumptions.

2. How This Part Handles It

In this part, the impact of environment differences is minimized.

The purpose is not to perform operations, but to observe behavior itself.
Therefore, in this part, a pre-configured environment is used as the assumption.

In this textbook, the **Killercoda** environment is used.

Killercoda is a hands-on environment for learning Kubernetes
and cloud-native technologies in a browser.

It can be started easily using a Google account or similar,
and the pre-configured environment can be used immediately
without installation or setup on a local PC.

In this textbook, it is used not to learn environment setup,
but as the foundation for observing Kubernetes behavior.

Cloud-native design is built on the assumptions that things break and that recovery is possible.

The main characteristics are as follows.

- Usable with only a browser
- No prior environment setup required
- Initial state is guaranteed
- Easy to start over after failure

This makes it possible to enter practice
with elements other than “what to learn” removed as much as possible.

3. The Role of the Environment

The environment used here is not one that closely resembles production, nor is it feature-rich.
It is one that reduces differences unrelated to the essence of learning.

The ability to restore immediately after breaking, and to try from the same state any number of times —
these properties allow focus on change and behavior.

However, in actual environments, the assumptions differ.

- Network configuration
- Storage
- Resource limits

These change significantly depending on the environment.

What is addressed here does not reproduce all of those.
It is simply the foundation for understanding structure and behavior.

4. Confirmation Before Entering Practice

What matters here is not the environment itself.
It is being conscious of what assumptions the observation is being made under.

Without this alignment of assumptions,
what is being observed becomes unclear.

With this, the assumptions for entering practice are in place.
Then, what actually happens within this environment?

Chapter 3: Hands-On Assignment - Experiencing Kubernetes Behavior Firsthand

1. Purpose of This Chapter

In this chapter, Kubernetes resources are actually handled
and their behavior is observed.

The purpose here
is not to memorize operations.

It is to experience firsthand what Kubernetes handles automatically,
and what the person no longer needs to do.

The goal is to see how the assumptions organized in Part 1
and the environment aligned in Part 2
actually appear as behavior.

2. How to Observe in This Chapter

In the hands-on work, the following perspectives are kept in mind.

Not “what did I do” but “what happened on its own”
Not “avoiding failure” but “how does it recover after breaking”

Kubernetes is a mechanism for reducing human operations.
In this chapter, by intentionally breaking and intentionally recovering, that design is confirmed.

3. Main Resources Appearing in This Chapter

In this chapter, several Kubernetes resources are handled.
There is no need to memorize the details here.

For now, simply grasp what role each one holds.

Resource	Role
Pod	The unit that executes a container
Deployment	The mechanism that maintains Pod state
Service	The mechanism that provides stable access to Pods
ConfigMap	The mechanism that manages configuration information
Secret	The mechanism that manages sensitive information
Volume	The mechanism that retains data
Ingress / Gateway	The mechanism that handles external connectivity

These are not independent functions.

Kubernetes keeps the system viable
by separating each role while combining them.

What matters is not memorizing the names.

Confirm through actual behavior why the roles are separated.

3-1. Pod / Deployment / Service - Separation of Execution Unit and Responsibility

■ Question
When units are separated,
who keeps them viable?

■ Operation (Overview)

- Create a Pod
- Delete a Pod
- Use a Deployment

- Change the replica count
- Access through a Service

■ Observation

- What happened when a Pod was deleted?
- What processing occurred that was not instructed?
- What is Deployment absorbing?

Pods are treated under the assumption that they break.

Kubernetes is not designed to prevent breakage —
it is designed to restore when something breaks.

3-2. ConfigMap / Secret / Volume - Separating Configuration from Execution

■ Question
Why is configuration separated from execution?

■ Operation (Overview)

- Create a ConfigMap
- Reference it from a Pod
- Delete the Pod
- Confirm the state after recreation

■ Observation

- What remains after the Pod disappears?
- Where is the configuration?
- Why is it not held within the Pod?

Pods are disposable.

In Kubernetes, the assumption is not long-running operation —
it is replacement.

3-3. Ingress / Gateway - Where Is the Boundary Placed?

■ Question
Where is external connectivity handled?

■ Observation

- The difference between Service and Ingress
- Why Ingress does not work on its own
- What is actually handling the processing

Kubernetes is a mechanism that defines not the method of communication,
but “how it is intended to be handled.”

3-4. kubectl - Used for Observation, Not Operation

In this chapter, the following commands are used.

- get
- describe

- logs
- exec

What matters is that in this textbook, kubectl is not used as a tool for operations.

- Viewing state
- Viewing change
- Viewing the gap

That is what it is used for.

4. Summary of This Chapter

What was done in this chapter is not memorizing operations.

It is confirming, as actual behavior,
how Kubernetes behaves under the assumption that things break.

- Pods disappear
- Deployments restore them
- Configuration moves outside
- Responsibilities are separated
- Communication is separated

All of these rest on the assumptions organized in Part 1.

kubectl is the window for viewing that state.

The experience up to this point was made viable on the Killercoda environment.
When the environment changes, the assumptions taken on also change.

Proceeding to the next chapter to confirm that.

3-1 Hands-On: Pod / Deployment / Service - Break, Restore, Delegate

Assumed environment: Killercoda (Kubernetes cluster running)
All subsequent operations use kubectl.

0. Pre-Check (Viewing the Current State)

First, confirm the state in which nothing has happened.

```
kubectl get pod
```

Observation

- In a state where nothing has been created, nothing exists
- **Kubernetes** “does not start anything on its own”

If nothing is defined, nothing happens

1. Create a Pod (A Single Execution Unit)

Create a Pod definition file.

```
cat <<EOF > pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: sample-pod
spec:
  containers:
  - name: nginx
    image: nginx:latest
    ports:
    - containerPort: 80
EOF
```

Create the Pod.

```
kubectl apply -f pod.yaml
```

Confirm the state.

```
kubectl get pod
```

Confirm the Pod details.

```
kubectl describe pod sample-pod
```

Observation

- A Pod is “a single execution unit”
- Kubernetes manages it but does not protect it

The Pod runs but is not maintained

※ **Note**

Kubernetes does not manage “what is running” —
it is a mechanism that operates based on “what the state should be.”

This Pod has no definition stating “it should continue to exist.”

Therefore, even if it is deleted, from Kubernetes’ s perspective
no “gap from the desired state” has occurred.

What has no defined state to protect is not maintained

2. Delete the Pod (Break It)

Delete the Pod.

```
kubectl delete pod sample-pod
```

Confirm the state again.

```
kubectl get pod
```

Observation

- The Pod is gone and does not return
- Kubernetes “does not protect a standalone Pod”

When it breaks, it ends

Question

- What would happen if this were production?
- Would a person recreate it every time?

3. Manage the Pod Using a Deployment

Create a Deployment definition file.

```
cat <<EOF > deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: sample-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: sample
  template:
    metadata:
      labels:
        app: sample
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80
EOF
```

Create the Deployment.

```
kubectl apply -f deployment.yaml
```

Confirm the Pod.

```
kubectl get pod
```

Delete the Pod. (Break it again)

```
kubectl delete pod -l app=sample
```

Confirm again immediately.

```
kubectl get pod
```

Observation

- The Pod is automatically recreated
- No instruction was given

State is maintained
What is protecting it is not a person but the mechanism

4. Change the Replica Count (Changing State)

Change the replica count of the Deployment.

```
kubectl scale deployment sample-deployment --replicas=3
```

Confirm the Pods.

```
kubectl get pod
```

Observation

- Three Pods exist
- Only the “count” is specified, not individual instances

The person specifies only the state

5. Access a Pod Through a Service

Create a Service.

```
kubectl expose deployment sample-deployment \
  --type=ClusterIP \
  --name=sample-service \
  --port=80
```

Confirm the Service.

```
kubectl get service
```

Confirm access through the Service.

```
kubectl run curl --rm -it --image=curlimages/curl --restart=Never -- \
  curl http://sample-service
```

Observation

- No Pod was specified directly
- The Service is in between

The entry point is handled, not the instance

6. Summary (What Was Confirmed in This Hands-On)

What was experienced in this hands-on is the following distinction.

Pod

- Execution unit
- Breaks
- Not protected

Deployment

- Management unit
- Maintains state
- Makes judgments in place of people

Service

- Connection point
- Conceals instances
- Separates responsibility

Kubernetes
is not a mechanism for running containers.

It is a mechanism for maintaining state.

Up to this point, the flow of break, restore, and delegate has been experienced.
Then, where is that state and configuration held?

3-2 Hands-On: ConfigMap / Secret / Volume - Separating Configuration from Execution

Assumed environment: Killercoda (Kubernetes cluster running)
All subsequent operations use kubectl.

0. Pre-Check

Confirm that the Deployment created in the previous hands-on exists.

```
kubectl get deployment
kubectl get pod
```

If Pods are running, that is sufficient.

1. Create a ConfigMap (Place Configuration Outside the Pod)

Define a ConfigMap.

```
cat <<EOF > configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
data:
  MESSAGE: "Hello from ConfigMap"
EOF
```

Create the ConfigMap.

```
kubectl apply -f configmap.yaml
```

Confirm.

```
kubectl get configmap
```

2. Reference the ConfigMap from a Pod

Configure it to read the ConfigMap as an environment variable.

```
cat <<EOF > deployment-config.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: sample-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: sample
  template:
    metadata:
      labels:
        app: sample
    spec:
```

```
      containers:
      - name: nginx
        image: nginx:latest
        env:
        - name: MESSAGE
          valueFrom:
            configMapKeyRef:
              name: app-config
              key: MESSAGE
        ports:
        - containerPort: 80
EOF
```

Apply it.

```
kubectl apply -f deployment-config.yaml
```

Confirm the Pod.

```
kubectl get pod
```

Confirm the environment variable.

```
kubectl exec -it $(kubectl get pod -l app=sample -o
    jsonpath='{.items[0].metadata.name}') -- env | grep MESSAGE
```

Observation

- The configuration is visible inside the Pod
- However, the definition is outside the Pod

Configuration is defined outside the Pod

3. Delete the Pod (Does the Configuration Remain?)

Delete the Pod.

```
kubectl delete pod -l app=sample
```

Confirm the Pod again.

```
kubectl get pod
```

Once the new Pod starts, confirm the environment variable again.

```
kubectl exec -it $(kubectl get pod -l app=sample -o
    jsonpath='{.items[0].metadata.name}') -- env | grep MESSAGE
```

Observation

- The Pod was replaced
- The configuration is still in use

The Pod does not "own" the configuration

4. Change the ConfigMap (Where Does the Configuration Take Effect?)

Change the ConfigMap.

```
kubectl edit configmap app-config
```

Change the content of MESSAGE and save.

```
MESSAGE: "Hello updated ConfigMap"
```

Restart the Pod.

```
kubectl delete pod -l app=sample
```

Confirm the environment variable again.

```
kubectl exec -it $(kubectl get pod -l app=sample -o
    jsonpath='{.items[0].metadata.name}') -- env | grep MESSAGE
```

Observation

- Configuration changes take effect when the Pod is recreated
- Configuration and execution are separated

Changing the configuration does not change the Pod definition

5. Use a Volume (Thinking About Where Data Is Placed)

Create a Deployment that uses a Volume.

```
cat <<EOF > deployment-volume.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: volume-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: volume-sample
  template:
    metadata:
      labels:
```

```
        app: volume-sample
    spec:
      containers:
      - name: app
        image: busybox
        command: ["/bin/sh", "-c"]
        args: ["echo hello > /data/message.txt && sleep 3600"]
        volumeMounts:
        - name: data-volume
          mountPath: /data
      volumes:
      - name: data-volume
        emptyDir: {}
EOF
```

Apply it.

```
kubectl apply -f deployment-volume.yaml
```

Confirm the data.

```
kubectl exec -it $(kubectl get pod -l app=volume-sample -o
    jsonpath='{.items[0].metadata.name}') -- cat /data/message.txt
```

6. Delete the Pod (Does the Data Remain?)

```
kubectl delete pod -l app=volume-sample
```

Once the new Pod starts, confirm again.

```
kubectl exec -it $(kubectl get pod -l app=volume-sample -o
    jsonpath='{.items[0].metadata.name}') -- ls /data
```

Observation

- The data is gone
- emptyDir shares its fate with the Pod

The lifetime of data is determined by the type of Volume

7. On Secrets (Not Practiced Here)

The concept of Secret is the same as ConfigMap.
However, because the use case differs, no operations are performed here.

What matters is not “whether it is kept secret” but “whether it is outside the Pod.”

8. Summary (What Was Confirmed in This Hands-On)

ConfigMap

- Places configuration outside the Pod
- Configuration remains even when the Pod disappears
- Configuration changes take effect when the Pod is recreated

Secret

- Places sensitive information outside the Pod
- Handled in the same way as ConfigMap

Volume

- Makes the lifetime of state explicit
- emptyDir shares its fate with the Pod

What was confirmed in this hands-on is that a Pod is an execution unit,
not a place for storing state or configuration.

Then, where is external connectivity handled?
Proceeding to Ingress / Gateway.

3-3 Hands-On: Gateway API and Access Confirmation - Where Does the Responsibility for External Entry Lie?

Assumed environment: Killercoda (Kubernetes cluster running)
All subsequent operations use kubectl.

0. Pre-Check

In the hands-on work up to this point,

- Deployment (nginx)
- Service (ClusterIP) These are assumed to exist. Proceed on that basis.

Confirm.

```
kubectl get deployment
kubectl get service
```

If sample-deployment and sample-service exist, that is sufficient.

1. Confirm the Role of Service (Internal Connection Point)

Confirm access through the Service.

```
kubectl run curl --rm -it --image=curlimages/curl --restart=Never -- \
  curl http://sample-service
```

Observation

- The access target is not a Pod
- Communication is possible with only the Service name
- The number of Pods and their replacement are not something to be aware of

Service is the internal connection point

2. Confirm the Assumptions of Gateway API (CRD and Controller)

Gateway API also does not work with the resource alone.
A Gateway Controller (implementation) is always required.

Confirm the CRD.

```
kubectl get crd | egrep
    "gateways.gateway.networking.k8s.io|httproutes.gateway.networking.k8s.io"
    || true
```

Confirm GatewayClass.

```
kubectl get gatewayclass
```

If GatewayClass is not present, the Controller may not be installed.
In that case, skip “3. - 5.” and reading “7. Responsibility Organization” is sufficient.

Declaration and execution are separated

3. Create a Gateway (Define the External Entry Point)

Confirm GatewayClass.

```
kubectl get gatewayclass
```

Confirm the available class name and replace .
(Examples: nginx / istio / envoy, etc.)

Create the Gateway.

```
cat <<EOF > gateway.yaml
apiVersion: gateway.networking.k8s.io/v1
kind: Gateway
metadata:
  name: sample-gateway
spec:
  gatewayClassName: <GATEWAY_CLASS_NAME>
  listeners:
  - name: http
    protocol: HTTP
    port: 80
    hostname: sample.local
```

```
EOF
```

Apply it.

```
kubectl apply -f gateway.yaml
```

Confirm the Gateway.

```
kubectl get gateway
kubectl describe gateway sample-gateway
```

Observation

- The Gateway defines the form of the entry point
- Where to route has not yet been decided

Gateway is a definition, not an execution

4. Create an HTTPRoute (Define the Flow)

Define an HTTPRoute resource.

```
cat <<EOF > httproute.yaml
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: sample-route
spec:
  parentRefs:
  - name: sample-gateway
  hostnames:
  - sample.local
  rules:
  - matches:
    - path:
        type: PathPrefix
        value: /
    backendRefs:
    - name: sample-service
      port: 80
EOF
```

Apply it.

```
kubectl apply -f httproute.yaml
```

Confirm the HTTPRoute.

```
kubectl get httproute
kubectl describe httproute sample-route
```

Observation

- The flow is defined on the HTTPRoute side
- The role is separated from the Gateway

Entry point and routing are separate

5. Access Through the Gateway

Find the Service associated with the Gateway.

```
kubectl get svc -A -l gateway.networking.k8s.io/gateway-name=sample-gateway ||
    true
```

Port-forward.

```
kubectl -n <NAMESPACE> port-forward svc/<SERVICE_NAME> 8080:80
```

Access from a separate terminal.

```
kubectl run curl --rm -it --image=curlimages/curl --restart=Never -- \
  curl -H "Host: sample.local" http://localhost:8080
```

If an nginx response is returned, it is successful.

Observation

- The Pod behind the Service is not being considered
- Processing proceeds through the flow Gateway → Route → Service

※ If the Service cannot be found or port-forward fails, the Gateway Controller may not be installed.
Even in that case, if “7. Responsibility Organization” is understood, the purpose of this section is achieved.

6. Delete the Pod (Does the Structure Collapse?)

Delete the Pod.

```
kubectl delete pod -l app=sample
```

Access again.

```
kubectl run curl --rm -it --image=curlimages/curl --restart=Never -- \
  curl -H "Host: sample.local" http://localhost:8080
```

Observation

- The Pod is replaced
- The access method does not change

The route is maintained

7. Responsibility Organization

Here, the role of each resource is organized.

Application (Pod)

- Performs processing
- Does not know the communication route

Service

- Handles internal connectivity

Gateway

- Defines the external entry point

HTTPRoute

- Defines the flow

The responsibility for communication lies outside the application.

8. What Was "Not Done" Is Important

In this hands-on, the following were not performed.

- Changing the application configuration
- Specifying IP addresses
- Being aware of Pod names

Even so, a state was achieved in which external access was possible and things continued to work even when Pods changed.
What does not need to be done reflects the intent of the design

9. Summary: What Was Confirmed in This Hands-On

Pod

- Focuses on processing

Service

- Handles internal connectivity

Gateway

- Defines the entry point

HTTPRoute

- Defines the flow

Communication is separated.
Proceeding to the next chapter with this structure intact.

3-4 kubectl Command Basics - A Tool for “Viewing State,” Not for Operating

In the hands-on work of this chapter, several kubectl commands are used.
There is no need to memorize all options and usage here.

What matters is not misunderstanding the role of kubectl.

The role of kubectl is not to directly operate and manage resources.
kubectl has commands such as delete and scale that change state.

However, in essence, it is a tool for confirming and understanding:

- What state is the cluster currently in?
- What is happening compared to the desired state?

In this chapter,

- get
- describe
- logs

are the focus, using kubectl in the way of observing state.

1. What Is kubectl?

kubectl is an interface for confirming cluster state
and communicating state.

kubectl itself does not make judgments or manage resources.

- Kubernetes makes the judgments
- kubectl only “communicates” and “views”

2. kubectl get

View the current state as a list

```
kubectl get pod
kubectl get deployment
kubectl get service
```

get is the command for confirming “what currently exists” as a list.

What can be understood here is:

- Whether it exists
- The count
- The rough state

Detailed reasons cannot be determined.
Confirm existence and state

3. kubectl describe

View the detailed state

```
kubectl describe pod sample-pod
```

describe is the command for confirming “why it is in this state right now.”

- Events
- Errors
- Scheduling results

If get is for confirming an overview of resources as a list,
describe is a tool for confirming the detailed state of individual resources.

Confirm the history and reason

4. kubectl apply

Communicate the desired state

```
kubectl apply -f deployment.yaml
```

apply is the command for communicating to Kubernetes the declaration “I want it to be in this state.”
What matters here is that it is not an operational command.

Kubernetes compares:

- The current state
- The declared state

and acts to close that gap.
Declare the desired state

5. kubectl delete

Break the state

```
kubectl delete pod sample-pod
```

delete is the command for deleting a resource.
However, in Kubernetes, deletion does not necessarily mean the end.

When a management unit such as a Deployment exists:

- The state breaks
- It is automatically restored

This is the behavior that occurs.
Break the state and observe the behavior

6. kubectl scale

Change only the count

```
kubectl scale deployment sample-deployment --replicas=3
```

scale is the command for changing only “how many should exist.”

- Which Pods to increase
- Where to place them

These are determined by Kubernetes.

Specify only the count

7. kubectl expose

Create a connection point

```
kubectl expose deployment sample-deployment --type=ClusterIP --port=80
```

expose is the command for exposing a resource through a Service.

Here too,

- Which Pod to connect to
- Processing when Pods increase or decrease

There is no need for a person to think about these.
The Service takes on that responsibility.

Define the connection point

8. kubectl logs / exec

Tools for “looking inside”

```
kubectl logs <pod-name>
kubectl exec -it <pod-name> -- /bin/sh
```

These are commands for confirming what is happening inside a container.

They are not for staying inside for troubleshooting.
They are tools for observation only.

Observe the inside

9. Summary

The correspondence to understand at this point is as follows.

Command	Role
get	View what currently exists
describe	View the detailed state
apply	Declare the desired state
delete	Break the state and observe behavior
scale	Change only the count
expose	Create a connection point
logs / exec	Observe the inside

kubectl is not a tool for operating Kubernetes.

It is a window for exchanging state.
It is sufficient to grasp only this relationship.

Proceeding to the next chapter.

Chapter 4: Points to Note When Deploying to a Local Environment - More Freedom Means More Responsibility

Through Chapter 3, using Killercoda for practice, the following were experienced firsthand.

- Basic behavior of Kubernetes
- Design under the assumption that things break
- A world where people do not operate directly

From here, the environment changes.

1. Differences Between Learning Environments and Local Environments

In a learning environment such as Killercoda:

- The initial state is in order
- Dependencies are aligned
- Unexpected gaps are few

These are the assumptions.

In a local environment, on the other hand:

- OS
- Container execution environment
- Network configuration
- Resource limits

All of these become environment-dependent.
This does not mean Kubernetes has become more difficult.

It simply means there is no longer anything absorbing the assumptions.

2. Points Where Environment Differences Become Visible

2-1. Network

In a local environment:

- Port-forwarding becomes necessary
- Name resolution does not work as expected
- External access is not possible

These problems arise.

This is not abnormal.

It occurs because the network boundary and DNS configuration
in a local environment differ from those in a learning environment.

2-2. Resources (CPU / Memory)

In a local environment:

- Insufficient memory
- CPU contention
- Forced process termination

These occur.

Containers go down, Pods restart.
This is not a Kubernetes problem.

In many cases, **Linux is dropping processes to protect resources.**
Kubernetes does not prevent that.

It operates under the assumption that this occurs.

These tend to occur in local environments, but can similarly occur in remote environments as well.
What matters is that this is not a Kubernetes problem — it is a property of the foundation, and is part of the assumption.

2-3. Storage

In a local environment:

- Volumes do not work as expected
- Data does not persist
- Permission issues block progress

These problems appear.

Here too, the same applies.
Kubernetes does not hold storage.

The actual dependencies are:

- OS
- File system
- Mounts

3. Freedom and Responsibility

In a local environment:

- Configuration can be set freely
- Structure can be decided independently

On the other hand, the reason when things do not work must also be taken on by the person themselves.
This is not difficulty.

It is simply that where responsibility lies has changed.

4. Connection to the Assumptions of Part 1

All the gaps encountered up to this point connect back to Part 1.

- Environments change
- Networks go down
- Resources run out
- People cannot control everything

The reason these surface in a local environment
is that the assumptions the learning environment was absorbing have been removed.

Kubernetes is not a mechanism for hiding those.
It is a mechanism for reducing the burden on people, under the assumption that those things occur.

5. Summary

When the environment changes, where responsibility lies also changes.
Which assumptions, and who is taking them on?

Whether one is conscious of that determines the design.

Proceeding from here into design with these assumptions in hand.

Part 2 Summary - Connecting What Was Experienced Firsthand to Design

What was done in this part was not operations.

By observing behavior,
it was confirmed how assumptions actually appear.

Units are separated, and the count increases.

As a result, state is not fixed —
it is treated as something that continues to change.

In this situation, breaking is not an exception —
recovery becomes the assumption.

Rather than people making each operation viable individually,
the desired state is defined and viability is left to the mechanism.

What becomes important here
is the boundary of what is delegated and what is taken on oneself.

How much is left to the mechanism, and where is responsibility retained?
The placement of that boundary becomes the design itself.

Then, on top of these assumptions, where are responsibilities divided?

Part 3: Application Step - Delivering and Connecting Applications

Introduction

In a microservices environment,
what flows through the system changes.

One is **change**.
Applications are updated and continuously delivered.

The other is **communication**.
Divided services connect and exchange with one another.

When units are separated, new flows emerge within the system.

What becomes a problem here is not the technology.
It is where to take responsibility for these flows.

First, the unit by which applications are divided is organized.

On top of that, how change is delivered to divided applications is addressed.
Furthermore, where the communication between divided services is controlled is addressed.

- Division
- Change
- Communication

These three are treated not as individual technologies,
but as a single structure.

What matters is not which tools to use.
It is the design of where to place that complexity.

Carry this perspective forward into the next chapter.

Chapter 1: Microservices and Application Configuration - What Happens When You Divide?

0. How Assumptions Are Set

This chapter does not address specific configurations or correct answers.

The way applications are divided changes depending on the assumptions in place.
What is addressed here is not which form is correct.

- **Why does the idea of division emerge?**
- **What happens when things are divided?**

That structure is what is addressed.

1. Why Does Division Appear?

What is addressed here is the application.

Processing that provides functions to users is configured as a single body.
However, as change increases, the situation changes.

- The frequency of changes increases
- The number of functions handled simultaneously increases
- The amount of processing exchange increases

What flows through the system continues to grow.

- **The flow of change**
- **The flow of communication**

In this state, a single change affects the whole,
and the whole must be handled every time a change occurs.

Even fixing just a part may require stopping the whole.
At this point, “continuing to handle things as a single body” becomes unrealistic.

What emerges here is the question: “Where do we draw the boundary?”

2. What Happens When Things Are Divided?

When an application is divided,
each part can be handled as an independent unit.

- Only a part can be changed
- Only a part can be run

However, at the same time, each part can no longer remain viable on its own.

When processing that was inside a single body is separated,
the need to connect what has been separated emerges.

One process calls another, one state is held in a different place,
and returning a result comes to span multiple units.

In other words, the moment of division is when **communication, state, and dependencies** emerge on the outside.

There are aspects that become easier to handle through division.

However, at the same time, a new problem emerges:
how to keep what has been divided viable together.

3. Complexity Moves

Division is not an act of reducing complexity.
It is an act of relocating complexity.

What was inside a single body emerges on the outside through division.

- Communication
- Change
- Dependencies

These come to be handled outside the divided units.

As a result, things that did not need to be considered before
now emerge as objects to be handled.

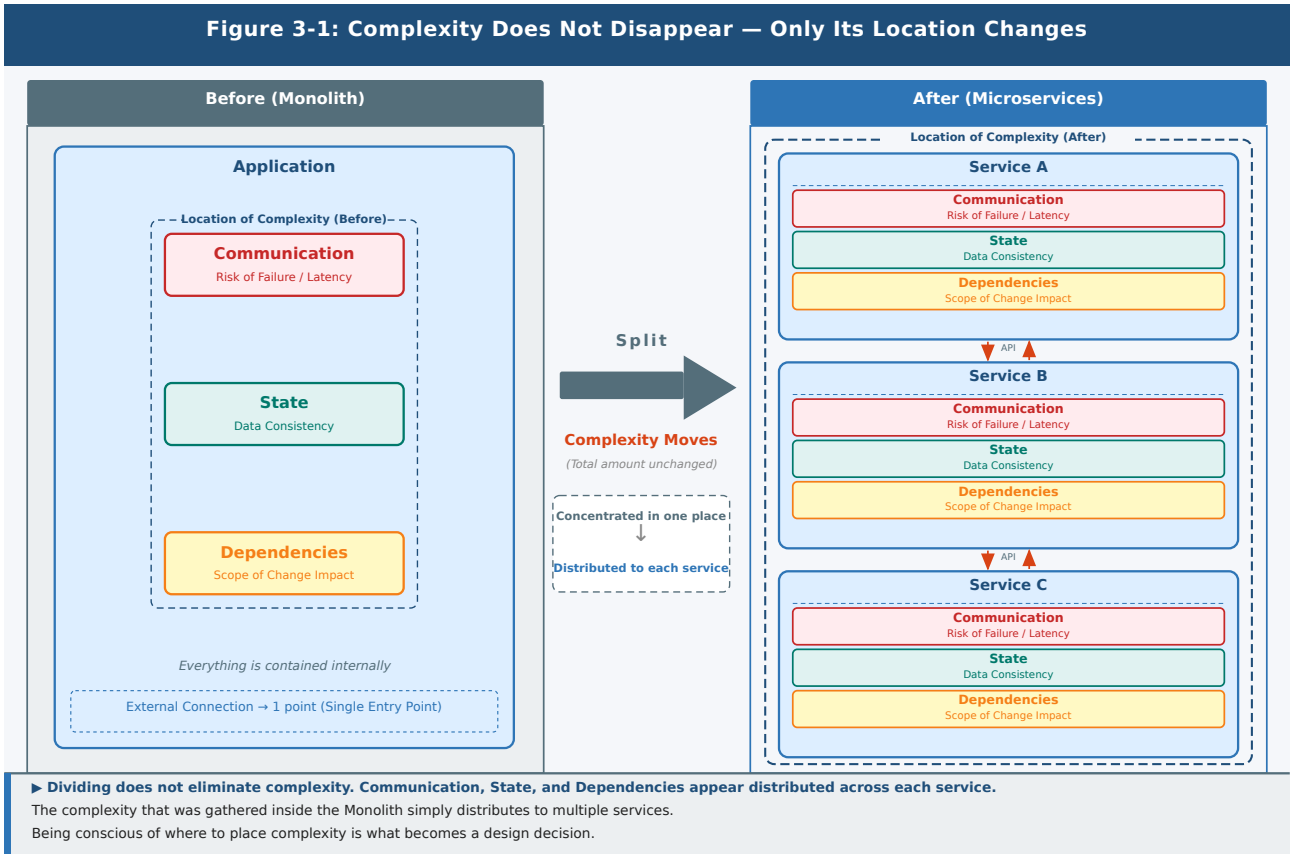


Figure 3-1: Complexity Does Not Disappear — Only Its Location Changes

- Communication is delayed and fails
- State drifts and inconsistencies arise
- Dependencies expand the scope of impact

These are not exceptions.
Division brings assumptions to the surface

4. Division as Responsibility

An application can be understood as a collection of responsibilities.

- Interaction with users
- Processing
- Data retention

The division itself is not what matters.

How far to treat as a single unit, and where to divide responsibility

This choice changes how things are handled.
Treating as a single unit means changes are made collectively.
Treating separately means a part can be handled independently.

However, at the same time, the need to connect what has been divided emerges.

The division changes what problems must be addressed.
This judgment becomes the design itself.

5. Division and Execution Environment

Divided units do not remain viable on their own.
Each runs in a different place, and each changes independently.

Some increase,
some decrease,
some fail,
and some are restarted elsewhere.

In order to maintain this state,

- Where to run
- When to run
- How to connect

judgments such as these constantly arise.

If all of this were handled by people, the state would need to be confirmed at every change and the necessary operations would need to be performed each time.

This state is not realistic

Therefore, a mechanism for continuously managing these things becomes necessary.
It is here, for the first time, that a mechanism such as Kubernetes takes on meaning.

Division reveals where responsibility must be placed

6. A Choice, Not a Correct Answer

It is here, for the first time, that the term “microservices” takes on meaning.

Microservices is not a form to aim for.
It is a design choice.

The optimal form changes depending on what assumptions are in place.
What matters is not division itself.

As a result of division:

- Which responsibilities are taken on
- Which changes are accepted
- How far control extends

That understanding is what matters.

7. The Question That Remains

Through division, the flows of change and communication have emerged.

Change is not something delivered once and finished.
It needs to be delivered repeatedly, safely, and continuously.

Then, how is that change delivered?

Chapter 2: Continuous Delivery and GitOps - Continuously Delivering Changes

1. Why "Deploy Once and Done" No Longer Works

In traditional systems, deployment was an event.
A release date was set, procedures were confirmed, and changes were applied.

This approach remained viable in environments where change frequency was low and the scope of impact was limited.
However, when applications are divided and the unit of change becomes smaller, this assumption breaks down.

Small changes occur frequently, and it becomes necessary to continuously track "who changed what, and when."
An increase in changes means a corresponding increase in the possibility that state will drift.

- Unintended changes become mixed in
- The state that is currently applied becomes unclear
- Returning to a previous state becomes impossible

Changes are not always applied correctly.
Change is also an uncertain flow

2. Continuous Delivery Is Not "Automation"

Continuous delivery is often equated with "automated deployment."
However, that is not its essence.

The problem is that every time a change occurs, a person must make a judgment,
confirm the procedure, and continue aligning state.

In this situation, the higher the frequency of change, the more unstable operations become.

Human operations fluctuate,
procedures shift subtly,
and results become unreproducible.

The harder people try, the more unstable things become

What Continuous Delivery aims for is:

- Being able to apply changes through the same procedure at any time
- Being able to reproduce state
- Being able to restore when a problem occurs

In other words, **creating a state that remains viable without requiring human judgment each time**

3. Why the Structure Becomes "People Do Not Deploy Directly"

In cloud-native environments,
a structure that does not make deployment dependent on human operations becomes the standard.

This is not because people are untrustworthy.

In an environment where changes have increased:

- Records of operations are not retained

- Procedures are not aligned
- State cannot be reproduced

These problems are unavoidable.

What matters is not “who did it” but “**what was done**”

Therefore, **a design is chosen that separates the act of deployment itself from human hands**.

4. The Idea of GitOps

One way of organizing this challenge is GitOps.

In GitOps, the desired state of the environment is defined in Git,
and a structure is taken in which the actual environment is continuously brought closer to that state.

What matters here is not the tool.
It is how change is handled.

- All changes are recorded in Git
- They can be compared as a gap
- It is possible to return to a previous state

A state is created in which the flow of change can be traced

Git is used because **it exists as a structure for managing change**

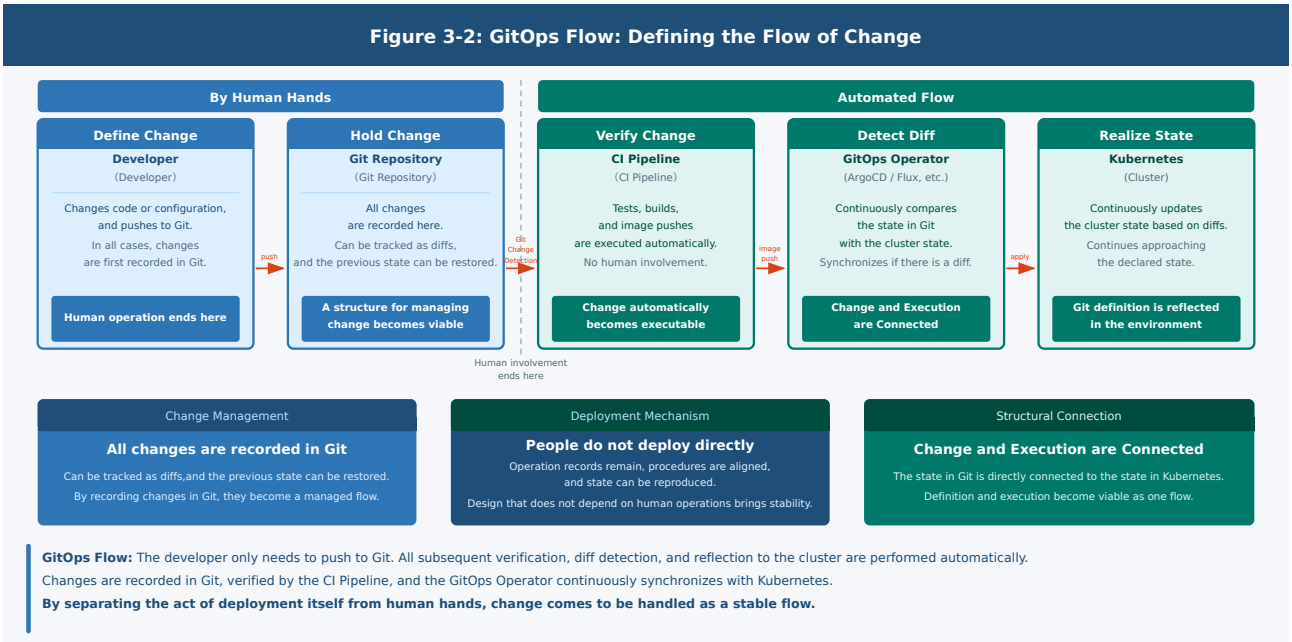


Figure 3-2: GitOps Flow — Defining the Flow of Change

5. The Relationship Between Kubernetes and GitOps

Kubernetes is a foundation on which the desired state can be defined declaratively.
The desired state is described, and the gap with the current state continues to be closed.

This property aligns with GitOps.

Kubernetes continuously brings the actual state closer to the state defined in Git.
Change and execution become connected

This structure is in fact also implemented as tooling.

ArgoCD and Flux compare the state in Git with the state of the cluster,
and keep this structure viable by continuously synchronizing the gap.

What matters is not the difference between tools.
The fact that this structure is viable

6. The Meaning of Continuous Delivery

As seen up to this point, in an environment where change has increased,
it is not realistic for people to continue making judgments each time.

Therefore, by defining state, fixing the flow of change,
and having a mechanism for applying it, **judgment is shifted to the mechanism**

7. The Question That Remains

What has been addressed up to this point is the flow of change.

Applications are updated,
state is rewritten, and it is reflected in the environment.

However, in a divided system, there is one more flow.
Services call one another and exchange processing.

It is the flow of communication.

The flow of change has been organized.
Then, how is this communication handled?

Chapter 3: Inter-Service Communication Control - Where to Take Responsibility for Complexity

1. The Moment of Division Is When Communication Becomes a "Design Problem"

Within a single application, communication is barely considered.
However, when an application is divided, this flow becomes communication over a network.

Network communication always carries the following assumptions.

- The possibility of not reaching its destination
- The possibility of latency occurring
- The possibility of the other side not responding

Communication carries the assumption of failure

From this moment, communication becomes an object to be treated as a design concern, not as an implementation detail.
When the behavior of communication is left to each service, behavior diverges and identifying causes becomes difficult.

Communication, if left unattended, becomes uncontrollable

Communication may succeed, fail, or be delayed.
The behavior of communication is a rule for how to act in situations such as success, failure, and latency.

2. Communication Control "Always Takes Place Somewhere"

The option of "not controlling" inter-service communication does not exist.

Judgments such as timeout, retry, and error handling
are always taking place somewhere, whether or not one is conscious of it.

The problem is where they are being taken on.

In the code of each service, in a shared library, or on the infrastructure side —
communication control already exists. It is simply not visible.

3. Complexity Does Not "Disappear" — It Moves

When communication control is taken on by the application side, behavior becomes clearer.
However, similar implementations multiply and unifying behavior becomes difficult.

When communication control is moved to the infrastructure side, code becomes simpler.
However, complexity appears in a different form: understanding configuration, visualizing behavior, and isolating failures.

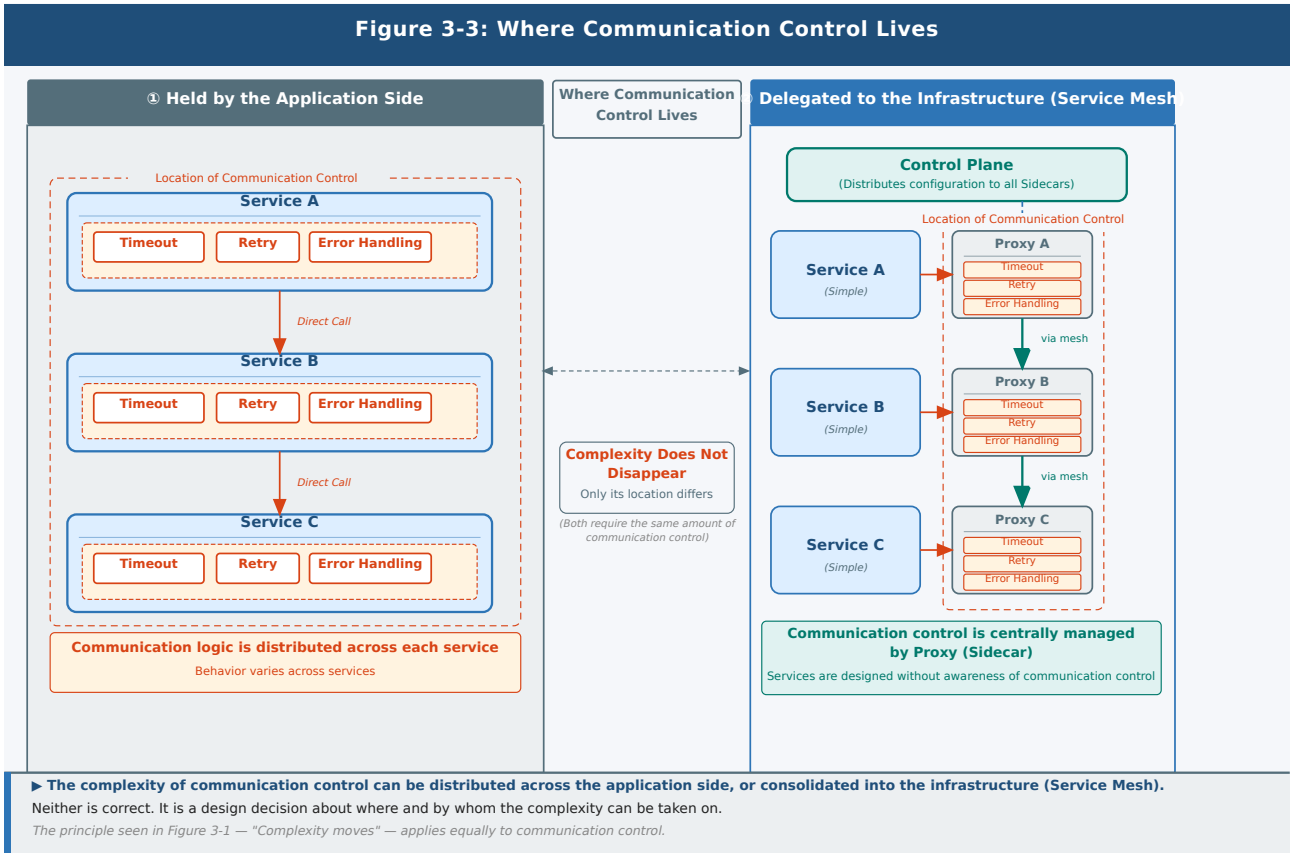


Figure 3-3: Where Communication Control Is Placed

Designing communication control is not eliminating complexity — it is deciding where to place it.
It is in this context that the idea of a **Service Mesh** emerges.

4. Not “Whether to Introduce” but “Whether It Can Be Taken On”

The question of this chapter is not “whether to introduce a Service Mesh.”
The essential question lies in the following points.

- Who understands this complexity of communication control?
- How far can it be traced when a failure occurs?
- Can it be taken on as an organization?

What is technically possible
and what can be taken on as an organization
do not necessarily align.

What cannot be understood cannot be operated

This structure is in fact also implemented as tooling.

Istio and Linkerd are implementations for moving communication control to the infrastructure side.
What matters is not the difference in functions.

The fact that communication control is separated out as a structure

5. The Question That Remains

What has been examined up to this point is the flow of communication.

Services are divided,
they call one another,
and exchange takes place over a network.

However, what is being addressed here
is only the structure of “where to control.”

**The behavior of communication is a rule for how to act
in situations such as success, failure, and latency.**

How is it defined,
how is it controlled,
and how is it observed?

Next, we look at how to handle
the behavior of communication itself.

Chapter 4: How to Handle the Behavior of Communication - Retry, Timeout, Observability

1. Communication “Appears as a Result”

In Chapter 3, where communication is controlled was examined.
However, what actually appears is not the configuration itself but its result.

Communication appears in the form of:

- Succeeding
- Failing
- Being delayed

Communication can only be observed as a result

2. Communication Without Defined Behavior Becomes Unstable

Communication carries the assumption of failure.

However, if it has not been decided what to do when failure occurs,
behavior will not be consistent.

- Some services retry
- Some services fail immediately
- Some services continue waiting for a long time

The same failure produces different results
This is a state in which communication is not controlled.

Communication without defined behavior becomes unstable

3. Behavior Is Defined as a "Rule"

The behavior of communication is defined as a rule.

- How long to wait
- How many times to try
- At what point to give up

All of these **decide how to act when failure occurs**
What is important here is that **it is behavior at the time of failure, not success, that determines the design**

4. Retry and Timeout

Among the behaviors of communication, the two most fundamental are:

- Retry
- Timeout

Retry re-attempts failed communication.
Timeout decides how long to wait.

Without these:

- Failure spreads immediately
- Continued waiting causes processing to back up

The system as a whole becomes unstable

5. What Cannot Be Seen Cannot Be Controlled

The moment communication leaves the system, it becomes invisible.

- Where is it slow?
- Where is it failing?
- What route is it taking?

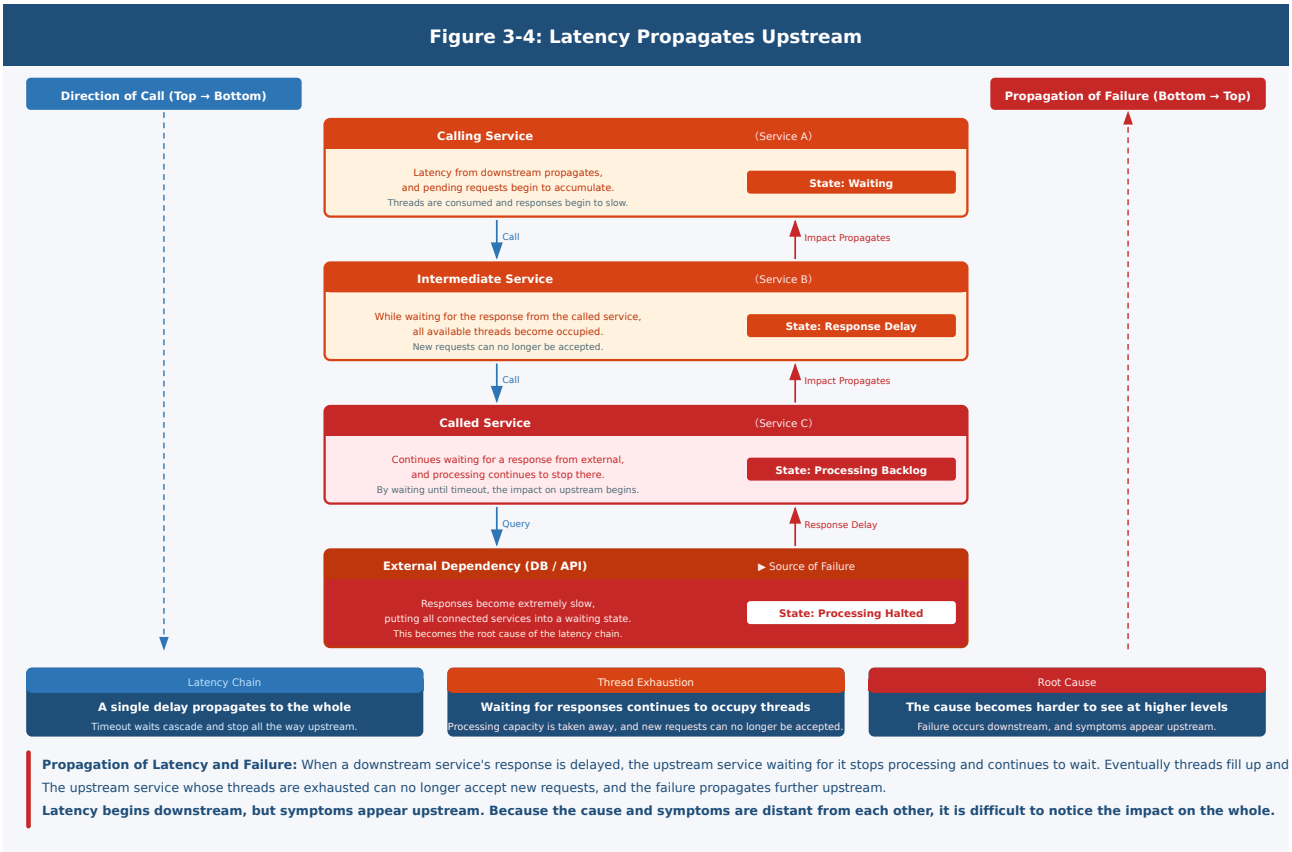


Figure 3-4: Latency Propagates Upstream

If these cannot be determined, the behavior cannot be adjusted.
What cannot be seen cannot be controlled

6. The Idea of Observability

In order to handle communication, it is necessary to understand state,
trace the flow, and identify the cause.

The idea for this is Observability.
Observability is not a tool.

It is the assumption required for understanding behavior

7. Where Is Behavior Taken On?

These behaviors can be:

- Held by the application
- Held by the infrastructure side

Neither is correct.

It is a design decision of where to take responsibility

8. Summary

What has been examined in this chapter is the behavior of communication.

Communication appears as the results of:

- Succeeding
- Failing
- Being delayed

What matters is whether it has been decided
how to act in response to those results.

Up to this point, the structure of:

- Division
- Change
- Communication
- Behavior

has come together.

Then, on top of these assumptions, how is the whole made viable?
This is organized in the Part 3 summary.

Part 3 Summary - What Was the “Judgment” Addressed in the Application Step?

In Part 3, the technologies that make up a cloud-native application were examined
not as methods or correct answers, but as an accumulation of judgments.

In a microservices environment, dividing applications creates two flows
within the system.

The flow of change, and
the flow of communication.

- How to divide
- How to deliver
- How to connect
- How to behave

In none of the chapters was a specific configuration or technology treated as “what should be chosen.”
This is because **what matters is not what is chosen, but whether one is conscious of what is being taken on**.

Technology Is Not an “Answer” — It Is a “Place to Take On Responsibility”

The technologies and ideas that appeared in Part 3 are not things that eliminate problems.

- Microservices are not something that reduces complexity — it is **a choice that relocates its placement**.
- GitOps is not something that eliminates judgment — it is **an idea that fixes judgment as a structure**.
- Inter-service communication control is not something that avoids the difficulty of communication — it is **a design that decides where to face it**.
- The behavior of communication is **a design as a rule** for how to act in response to its results.

Technology does not make judgment unnecessary.

It is a means of expressing as structure where judgment is taken on

The Question That Still Remains

Divide, deliver, connect.

Even having organized all of this, systems do not automatically become safe.
By moving judgment outside as a structure, the next question becomes clearer.

What is happening in the system right now?
Is that judgment functioning correctly?

Then, how can it be observed?

Part 4: Operations Step - Making Things Visible and Protecting Them

In Part 3, the structure for dividing, delivering, and connecting applications was organized.

However, merely moving judgment outside as a structure
is not enough to know whether the system is functioning correctly.

In a distributed system, the internal state is not directly visible.
What is visible is only the behavior that appears as a result.

- Response is slow
- Errors are returned
- Processing stops

Such phenomena can be observed, but where the cause lies cannot be determined without further investigation.

- Is it a problem in the application?
- Is it a problem in the network?
- Is it a problem in the database?

Without the ability to make that judgment, response is delayed.
Without knowing the cause, the same failure repeats.

In this state, stable operations cannot remain viable.

Even when the structure is in order, if the state cannot be determined, the whole cannot be maintained.
What is needed here is not the ability to directly look inside.

It is the perspective of inferring state from information that can be observed.

What is happening in the system right now?
And where does that judgment come from?

Chapter 1: Monitoring and Observability - Visibility Alone Is Not Enough for Judgment

1. The Moment of Distribution Is When "The Whole Becomes Invisible"

In a single application:

- State is gathered in one place
- Logs provide answers
- Metrics allow understanding

However, this is viable only because everything is operating as a single body.

When an application is divided:

- Changes can be made flexibly
- Only a part can be updated
- Scaling can be done independently

At the same time, state becomes distributed.

Multiple services operate, and each holds state independently.
Some processing spans multiple services, and results are returned over a network.

From this moment, the state of the system can no longer be grasped in one place.

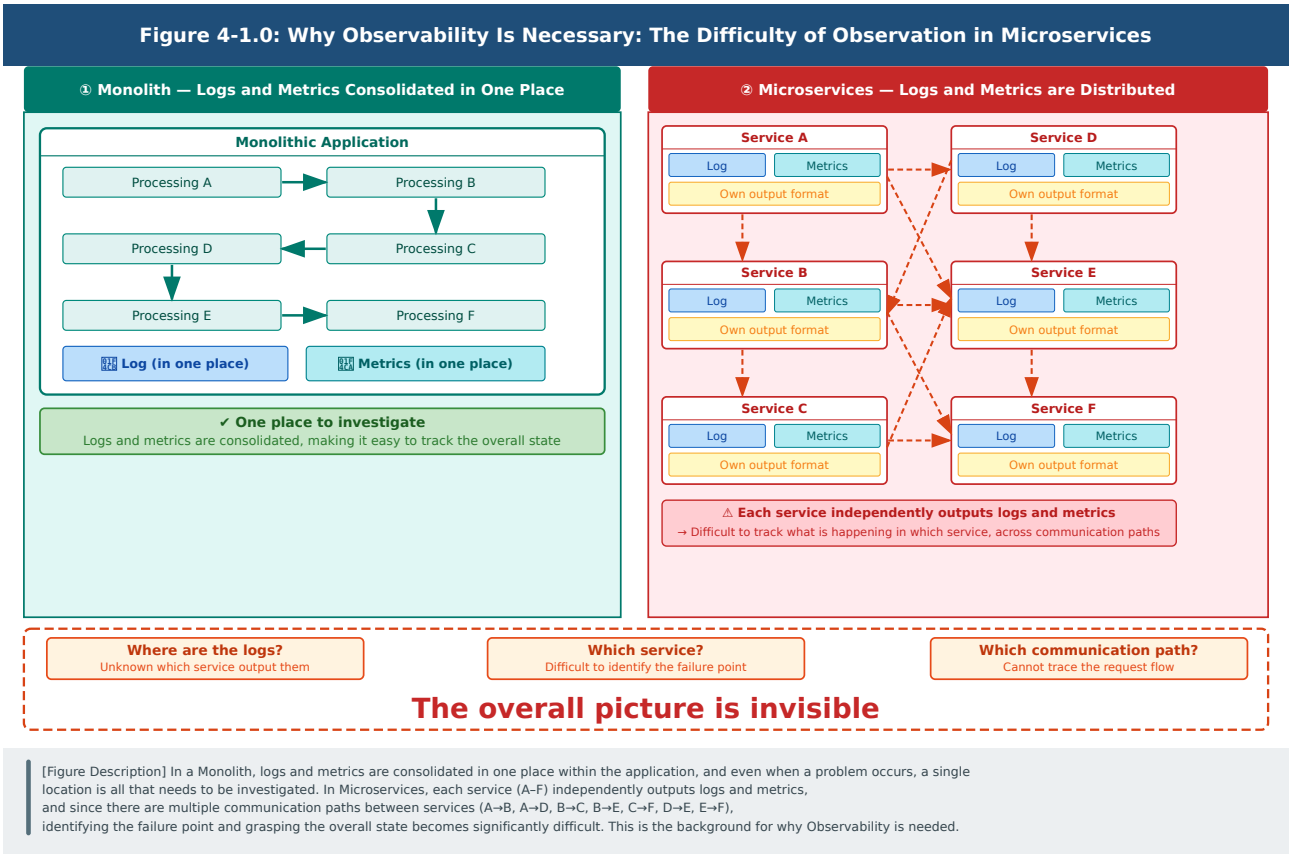


Figure 4-1.0: The Moment Things Become Distributed, the Whole Becomes Invisible

2. The State of “Visible but Not Understandable”

Even in a distributed system:

- Information exists
- Logs are being output
- Metrics can be obtained

However, those are fragmented.

This is because each service operates independently
and outputs information in different places at different times.

- An error is occurring in one service
- A delay is occurring in another service
- Resources are under pressure in yet another place

Each can be observed,
but how they connect as a single problem is not clear.

3. Observation Is Possible, but Judgment Is Not

Having information and being able to make judgments are different things.

Even with a large volume of logs, the cause cannot be identified.
Even with complete metrics, where the problem lies cannot be determined.

Fragmented information carries no meaning on its own.

Only by connecting how those things relate to one another
does the state of the whole become visible.

4. From "Seeing" to "Judging"

What is needed here is not an increase in information.

What matters is
whether state can be inferred from observed information.

- Where is latency occurring?
- Which processing is the bottleneck?
- Which change is having an impact?

Only by being able to make these judgments
does stable operations first become viable.

5. What Is Observability?

Observability is not seeing everything.

It is the property of being structured so that the internal state of a system
can be inferred from observable information such as logs, metrics, and traces.

It is not a matter of collecting all information —
it is a matter of being in a state where information necessary for judgment can be handled.

In real operations, observation also has a cost.

- Too many logs and they become unreadable
- Too many metrics and they become unmanageable
- Too many traces and system load increases

Therefore, Observability is also
a design for deciding **how far to observe**.

6. The Difference from Monitoring

Monitoring watches pre-determined indicators.

- CPU utilization
- Response time
- Error rate

It is possible to detect signs of abnormality.
However, it cannot respond to problems that were not anticipated.

Observability is the assumption required for inferring state from observed information.
Tools are merely means that support this observation.

- Prometheus collects metrics
- Grafana visualizes them
- Cilium observes communication state

What matters is not the tools.
It is the judgment of what to observe.

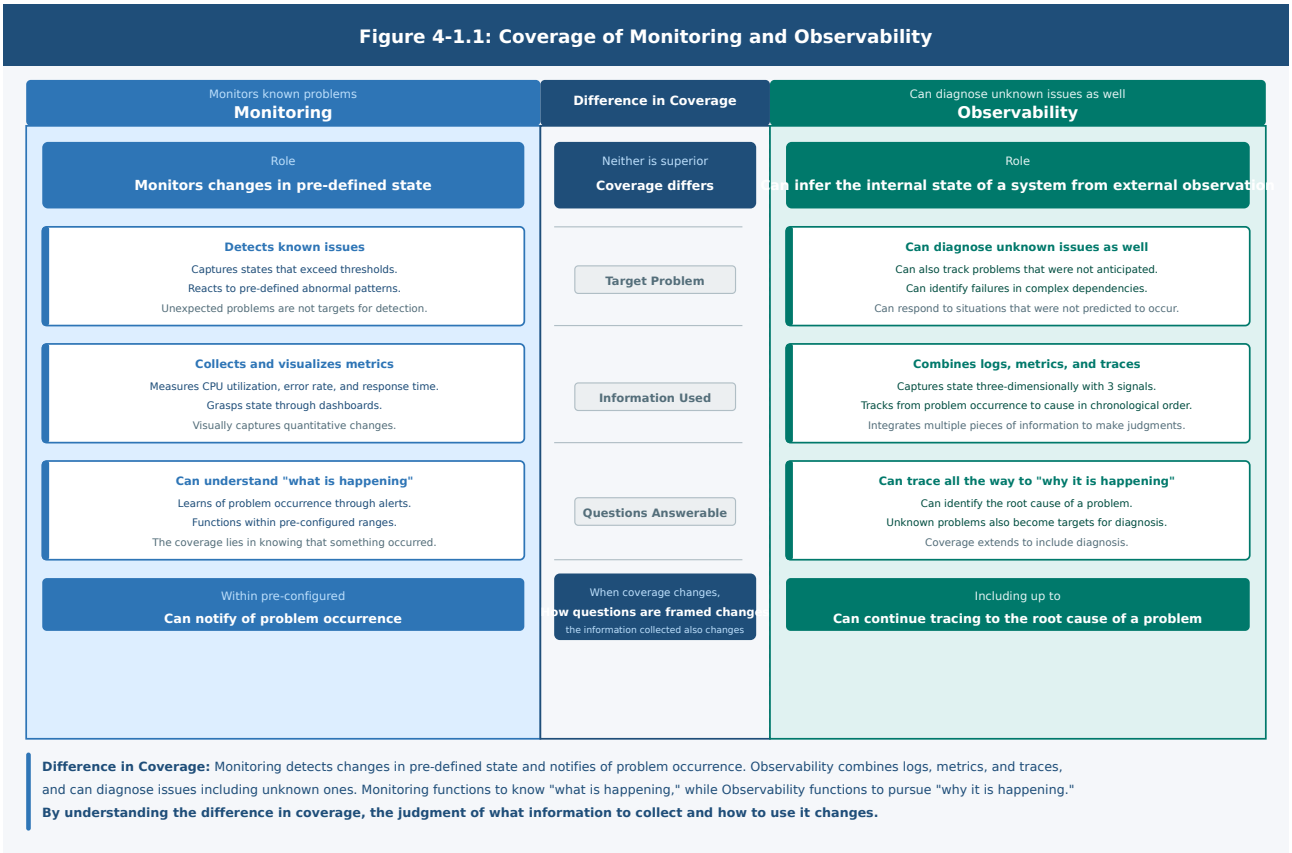


Figure 4-1.1: Coverage of Monitoring and Observability

7. The Question That Remains

Once observation becomes possible, is that sufficient?

- Which information should be looked at?
- How should they be related?
- How should abnormalities be judged?

How do those become a single judgment?

Chapter 2: Tracing and Visualization of Distributed Systems

1. Problems That Occur in Distributed Systems

What is addressed here is an application divided into multiple services.
Each service operates independently, and together they make a single function viable.

In a microservices environment, a single request is processed through multiple services.

- An API service receives the request,
- An authentication service performs authentication,
- A data service accesses the database.

For example, in an e-commerce site:

- An order service accepts order information,
- A payment service handles payment processing,
- A notification service reports the result to the user.

Even though it appears to the user as a single purchase operation, internally multiple services are coordinating to advance the processing.

In this structure, processing does not complete in one place.

- Where is processing slowing down is unknown
- Which service is experiencing an error is unknown

Each service can be observed.

However, how they connect as a single request is not clear.

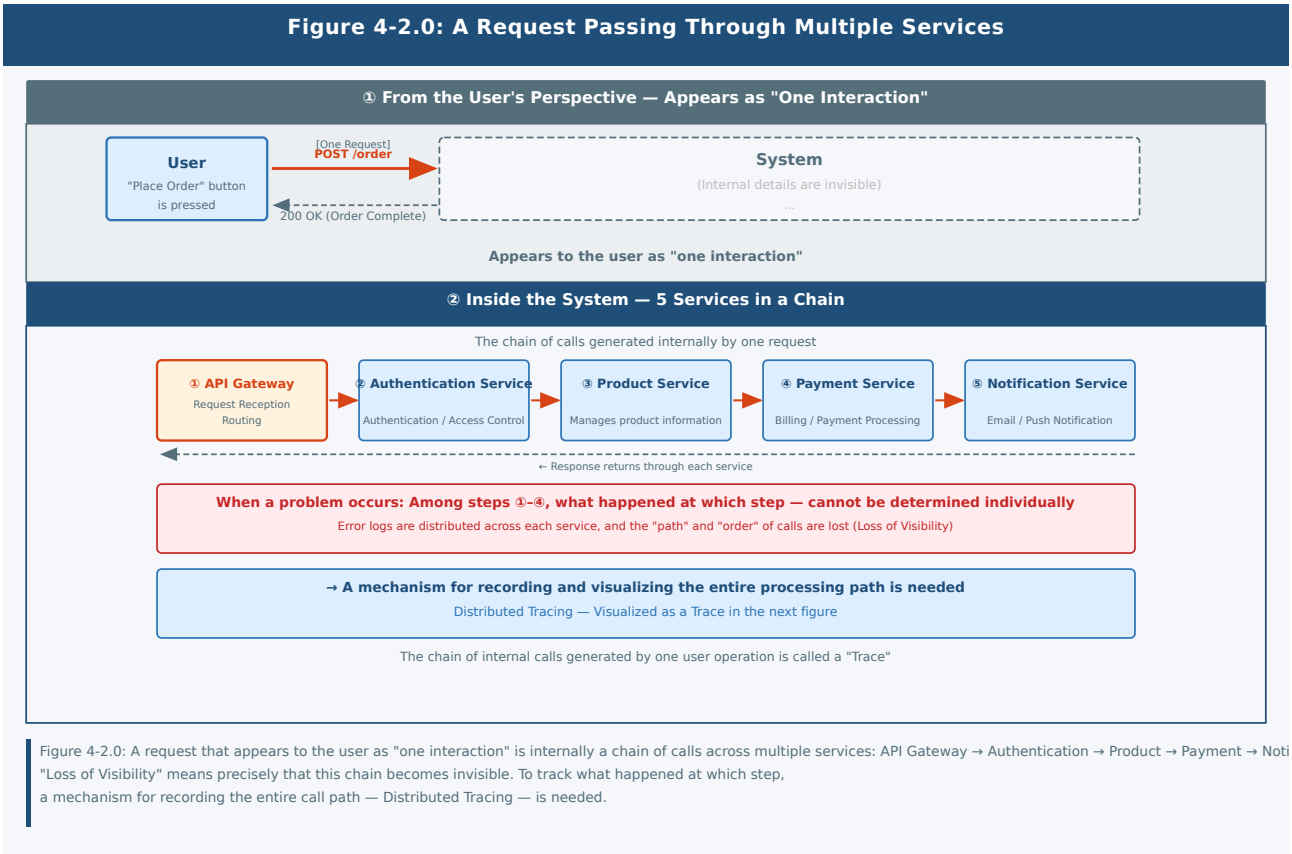


Figure 4-2.0: A request that appears to the user as "one interaction" is internally a chain of calls across multiple services: API Gateway → Authentication → Product → Payment → Noti
"Loss of Visibility" means precisely that this chain becomes invisible. To track what happened at which step,
a mechanism for recording the entire call path — Distributed Tracing — is needed.

Figure 4-2.0: A Request Passing Through Multiple Services

In this state:

- Even when a problem occurs, the cause cannot be identified
- Even when a response is taken, reproducibility cannot be maintained
- The same failure repeats

Without understanding the flow of processing, the system cannot be operated stably

2. The Idea of Distributed Tracing

What addresses this problem is distributed tracing.

It makes it possible to track how a single request
was processed within the system.

- Which services the request passed through
- How long each service took

By treating these as a single flow,
the whole of the processing is visualized.

- Where latency is occurring
- Where errors are occurring

These can be identified at the level of a single request.
However, it does not mean that every request should be traced.

Data volume continues to grow, storage costs increase,
and system load is also affected.

Tracing is not about tracing everything —
it is a design for deciding how far to trace

3. The Structure of Trace Data

In distributed tracing,
processing information for requests is recorded as traces.

A Trace represents the overall flow of a single request,
and a Span represents the processing in each service.

By recording the processing that occurred in each service as Spans
and associating them as a single Trace,

processing that was fragmented
is reconstructed as the flow of a single request.

4. The Role of Representative Tools

■ Jaeger
Jaeger is a tool for collecting and storing trace data
and visualizing the flow of processing.

■ OpenTelemetry
OpenTelemetry is a standard specification
for handling observation data such as logs, metrics, and traces in a unified way.

What matters is not the tools.

- **Which processing to trace**
- **At what granularity to observe**

That judgment is what matters.

5. The Question That Remains

Is it sufficient to be able to trace the flow of processing?

- Which requests to trace
- At what granularity to record

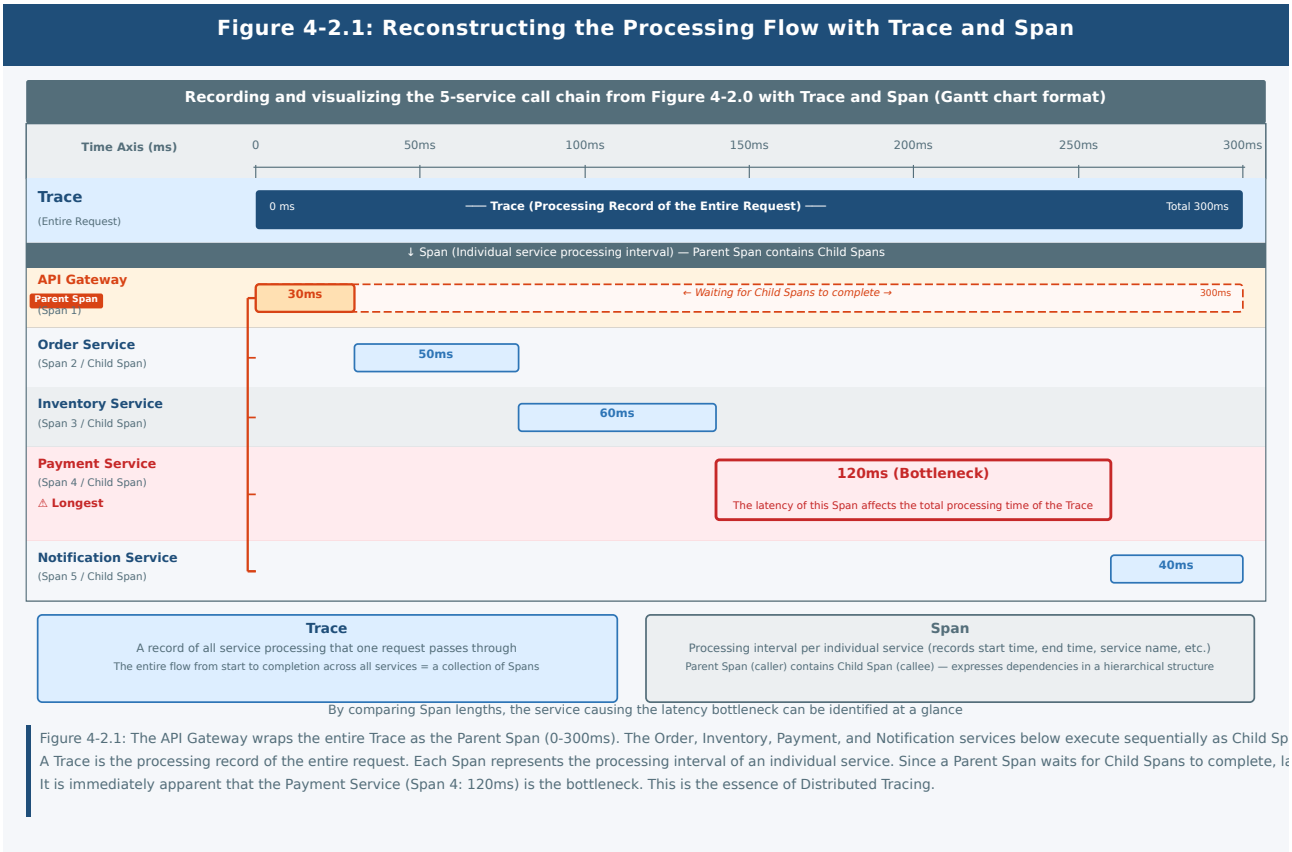


Figure 4-2.1: Reconstructing the flow of processing with Trace and Span

- Which information to retain

How does the design of observation become viable?

Chapter 3: Security and Access Control - What to Restrict to Prevent Breakage

1. The Moment of Distribution Is When "Places That Can Be Broken" Increase

What is addressed here is a system divided into multiple services.

Security here does not mean only protecting against attacks.
It also includes the meaning of controlling operations so as not to break the system.

As applications are divided, services increase,
APIs increase, and communication routes increase.

In this structure, points at which the system can be operated continue to increase.

- Which services can be accessed?
- Which resources can be operated?
- Which configurations can be changed?

Without appropriately controlling these, unintended operations are performed,
impact reaches areas that should not be touched, and information leakage occurs.

2. The Risk of "Being Breakable"

In a cloud-native environment:

- Deployment
- Configuration changes
- Resource operations

These operations can be performed flexibly.

While this is a powerful characteristic,
it is also the ability to break the system.

Even beginners can handle strong permissions, so unintended changes or deletions
may impact the system as a whole.

In this state, it becomes ambiguous who can operate how far,
and a single operation impacts the whole.

The system cannot be operated stably.

Security is not only for protection against attacks.
It is a mechanism for preventing accidents and controlling the scope of impact.

3. Access Control (RBAC)

What becomes necessary to address this problem is access control.

In Kubernetes, RBAC (Role Based Access Control) is used
to define operational permissions for users and services.

The purpose of RBAC is not to grant permissions.

It lies in limiting how far operations can be performed and controlling the scope of impact.
It is not a design that increases "what can be done" — **it is a design that restricts** "**the range that can be broken.**"

4. Policy Management

Access control alone is insufficient.

- What operations are permitted?
- What configurations are allowed?

These must be managed as consistent rules.

As a mechanism for such policy management,
OPA (Open Policy Agent) is used.

The purpose of policy is not to increase rules.

It lies in preventing unordered operations and maintaining consistent control.
In a flexible environment such as Kubernetes, without governance things easily break down.

5. Secrets Management

The handling of sensitive information is the same.

- Database passwords
- API keys

- Authentication tokens

These are not mere configuration.

If they are leaked, the impact extends to the system as a whole.

- From where can they be referenced?
- Where can they be used?

That range must be controlled.

6. The Question That Remains

Is it sufficient to be able to control?

- How far to restrict?
- At what granularity to manage?
- What range to permit?

How is that design determined?

Chapter 4: Operations and Troubleshooting - A Design for Continuing to Make Judgments

1. Operations Means "Keeping Things Going"

A system does not end once it is running.
It must continuously provide services.

However, in a cloud-native environment,
containers are dynamically created and deleted,
services are distributed, and configuration continues to change.

In this state:

- Everything cannot be grasped
- Everything cannot be continuously tracked

Even so, services cannot be stopped.

- Where is the problem?
- How far has the impact spread?
- How far should a response go?

These must be judged each time they occur.

Operations is not about knowing everything.
It is about continuing to make judgments in order to keep things going, based on limited information.

2. What Is Being Done in Failure Response?

Failures cannot be completely prevented.
What matters is being able to respond appropriately when they occur.

In general, failure response proceeds in the following flow:

- Detect

- Grasp the situation
- Identify the cause
- Restore
- Prevent recurrence

However, in actual operations, it is not always possible to proceed in order.
What is needed first is to restore the service.

- Whether to investigate deeply
- Whether to temporarily work around

A judgment must be made on how far to respond.

However, this judgment is made in a state of incomplete information.

- The cause is unknown
- It cannot be reproduced
- The same failure repeats

Without the ability to make a judgment, operations cannot remain viable.

3. Why Tools Become Necessary

For this problem,
it is not possible for people to continue directly tracking state.

- Entering a Pod to continue investigating
- Manually changing state

Such operations can resolve things temporarily,
but they break the structure of the system and cause problems over the long term.

Kubernetes is a mechanism that manages declaratively.

Ad hoc operations continue to generate drift from the original state.

Furthermore:

- Distributed services must be understood across the board
- Past state and changes must be tracked
- Multiple pieces of information must be combined to make judgments

kubectl alone cannot respond to this state.

Therefore,

- Metrics
- Logs
- Tracing

These pieces of information need a mechanism that handles them in an integrated manner.

What matters is not the tools.
It is being in a state where information necessary for judgment can be handled.

4. The Assumption of Continuing to Improve

Failure response does not end with restoration.

- Why did it occur?
- How can it be prevented?
- Which observation was insufficient?

These must be reviewed.

However, what matters is not making everything perfect.

- How far to respond?
- Where to accept limitations?

By repeatedly making these judgments,
the stability of the system gradually increases.

5. Design for Reliability (SRE)

The idea that supports this judgment is SRE.

In SRE:

- SLI (indicators)
- SLO (objectives)

These are defined, and:

- How far to maintain quality?
- How much failure to accept?

is decided.

Reliability is a state in which quality is visualized as a number
and a judgment can be made on how far to respond.

Not everything can be protected.
That is precisely why it is necessary to decide what to protect.

6. Summary

Operations is not about grasping everything.

It is about continuing to make judgments based on limited information
within a system that is distributed and continues to change.

State is understood through observation,
impact is limited through control,
and on top of that, how far to respond is decided.

Not everything can be prevented.
Not everything can be understood.

That is precisely why:

- How far to look
- How far to control

- How far to accept

these judgments become necessary.

Technology exists to support those judgments.

Part 4 Summary

In Part 4, the thinking required to make cloud-native systems visible and operate them safely was organized.

The essence is a single point. **Cloud-native operations is "a continuous series of judgments."**

- Observability is not about observing everything —
  it is about deciding how far to observe.
- Tracing is not about tracing everything —
  it is about deciding how far to trace.
- Security is not only for protecting against attacks —
  it is a mechanism for preventing the system from being broken.
- Operations is not about investigating everything —
  it is about deciding how far to respond.

See, trace, control, judge.
These are connected as a single sequence.

In a cloud-native environment, systems continue to change at all times.
It is not possible to do everything perfectly.

That is precisely why the ability to decide how far to go becomes important.

This judgment is not mere selection.

In the midst of incomplete information,
it is the act of deciding how far to accept, how far to control, and how far to tolerate.

And the accumulation of those judgments determines the behavior of the system,
and directly affects value, cost, and even the way business is conducted.

Part 5: Expansion Step - Turning Technology into Value

In Parts 1 through 4, the cloud-native system was examined
not as a collection of technologies, but as a structure.

Units were separated,
state was controlled,
change was observed,
and where responsibility lies was organized.

What has been examined up to this point is the structure for keeping a system viable.
However, it is not sufficient for a system to simply remain viable.

That structure — for whom is it used?
What does it support, what does it change, and what value does it connect to?

Here, a new question emerges.

How does this structure generate value?

The value addressed here is not limited to revenue.
It also includes the very fact that the system continues to function.

In Part 5, the meaning that the cloud-native structure examined so far
holds within society and business is organized.

What is addressed is not specific tools or implementation methods.
It is how the structure connects to the provision of value and decision-making.

Carry this perspective forward into the next chapter.

Chapter 1: Social Challenges and the Resolving Power of Cloud Native

1. The Fact That the Assumptions of Society Have Changed

Cloud native did not spread because it is “convenient technology.”
It emerged **as a response to the fact that systems could no longer remain viable under the traditional assumptions**.

Systems of the past were designed under the assumption of “being stable.”
Things once built were run for a long time, and when problems occurred, people responded.

This approach remained viable in situations where the environment did not change significantly and failures could be treated as exceptions.

However, today, these assumptions have broken down.

Services change continuously,
usage patterns are unpredictable,
systems are composed of multiple elements,
and they become a distributed structure that assumes a network.

The fact that change, distribution, and uncertainty have become assumptions is what this represents.

2. "Continuing to Remain Viable" Has Become More Difficult Than "Building"

This change has significantly altered what is difficult about systems.
Previously, "**how to build**" was the central problem.

Today, "**how to continue to remain viable**" has become the central problem.

- Parts break
- State drifts
- Environments change

These are not abnormal — they are assumptions.

Under these assumptions:

- Manual operations cannot keep up
- Fixed configurations cannot be maintained
- Integrated systems cannot withstand change

In this state, a system cannot continue to remain viable for long.

Furthermore, as real constraints:

- Personnel cannot be increased
- Costs are limited
- Available time for response is also limited

These conditions accumulate.
Not only does it not remain viable — it also cannot be maintained.

3. Why Structure Becomes Necessary

For this situation, it is not possible to resolve things through human response alone.

- Change cannot be stopped
- Distribution cannot be avoided
- Uncertainty cannot be eliminated

Under these conditions, what is needed is not stability,
but a structure that continues to recover even while breaking.

This is the reason why mechanisms such as:

- State Management
- Self-Healing
- Scaling
- Rescheduling

become necessary.

4. The Problem That Cloud Native Is Solving

Cloud native is an answer to the challenge of creating "a structure that continues to remain viable."
Everything examined in Parts 1 through 4 exists for this purpose.

- Separating execution units
- Delegating control to the mechanism

- Choosing the assumptions of the environment
- Taking uncertainty as an assumption

These are not individual technologies.

They are a structure for accepting change, taking distribution as an assumption,
and continuing to remain viable amid uncertainty.

5. Understanding It as Structure, Not as Technology

When cloud native is viewed as a collection of tools:

- Learning Kubernetes
- Using cloud services
- Handling containers

the understanding stops there.

However, the essence is not that.

- What to separate, and where to hold responsibility
- How far the application holds
- How far to delegate to the infrastructure
- Where to manage state

This choice of boundaries becomes the standard for all design.

Cloud native is not “a better method.”
It is the structure chosen in situations where, without it, things cannot remain viable.

6. The Question That Remains

Through structure, the system has come to continue to remain viable.

Then, that structure —

- How does it connect to value?
- How does it influence decision-making?
- How does it function within society?

Next, how this structure appears as value will be examined.

Chapter 2: Shifting from SES to a Value-Delivery Model

1. Decomposition Changes How Responsibility Is Divided

- Through containers, applications were separated as execution units.
- Through Kubernetes, control of those units came to be delegated to the mechanism.
- Through cloud, part of the infrastructure came to be left to the outside.

This is not merely a change in technology.

It is a change in which **the units that make up a system are decomposed,
and responsibility is allocated to each**.

Decomposition is not making a structure more fine-grained.
It is **redistributing where responsibility is placed**.

Through this change, the assumption of "who handles how far" changes.
As a result, the very way work is conducted changes.

2. Value Is Determined by "How Far Responsibility Is Held"

Even when handling the same technology:

- Performing instructed work
- Providing an environment
- Taking on operations
- Holding responsibility for results

the value provided is entirely different.
This difference is not a difference in skills.

It is determined by **how far one treats as within their own scope of responsibility**.

3. Why Work Can No Longer Be Handled as Tasks

In an environment where change, distribution, and uncertainty are assumptions,
it becomes impossible to respond simply by carving out task units and allocating them to people.

This is because in a decomposed structure:

- Looking at only a part does not reveal the whole
- Processing is distributed, and cause and effect are separated
- State continues to change, and assumptions cannot be fixed

These situations constantly arise.

In this state:

- Problems cannot be identified
- The same failure repeats
- Where responsibility lies becomes ambiguous

As a result, **simply completing tasks means the system cannot continue to remain viable.**

4. From "What to Do" to "What to Keep Viable"

What is required in this situation is not tasks.
It is observing state, identifying problems, and continuing to improve.

In other words, what is asked is not "**what to do**" but "**what to keep viable**" .

"Viability" here means:

- State can be grasped
- Problems can be identified
- Where responsibility lies is clear

A state in which these conditions are met.

The observation, control, and judgment addressed in Part 4
are the assumptions that support this viability.

5. Structure Determines Value

When structure changes, how value is produced changes.

In a monolithic structure,
treating as a single unit is the assumption.

In a distributed structure, on the other hand:

- Separated units
- Control by the mechanism
- Design that takes change as an assumption

Judgments are made under these.

Within this:
How far to hold responsibility
How far to keep viable

This choice becomes value itself.

Cloud native is not “a better method.”
It is a structure that changes the very way responsibility is held.

6. The Question That Remains

When the scope of responsibility determines value:

- How is that responsibility defined?
- How far should it be taken on?
- How is it kept viable as an organization?

Chapter 3: FinOps - Cost Is Determined by Structure

1. Why Cost Problems Appear

In a monolithic structure, the system is handled as a single body.
Resource usage was relatively fixed, and costs were predictable.

However, in a cloud-native environment,
the system is distributed,
configuration changes dynamically,
and resources continue to increase and decrease.

Through this change, **cost becomes not something fixed but something that always changes.**

2. What Cannot Be Observed Cannot Be Controlled

The first problem that arises in this situation is a state in which it is not known what costs how much.

As seen in Part 4, **what cannot be observed cannot be controlled.**
Cost is the same.

Furthermore, when it becomes ambiguous who holds responsibility for that cost:

- Optimization does not take place
- Waste is left unaddressed
- Cost continues to increase

As a result, **cost continues to grow without being controlled.**

3. What FinOps Is Addressing

The organization for this problem is FinOps.
FinOps is neither a tool nor a method.

It is a structure for observing cost, dividing responsibility, and continuously optimizing.

- Visualization (what costs how much)
- Responsibility Boundary (who holds that cost)
- Continuous optimization (under the assumption of continuing to change)

This directly corresponds to the structure examined in Parts 1 through 4.

4. Cost Appears in Design

Cost appears as a result of design.

- At what unit to divide
- How much to scale
- How far to automate
- Which environment to choose

These choices are directly reflected in cost.
Cost is “the shadow of design.”

5. Cost Constrains Design

Cost is not merely a result.

- Which configurations can be chosen
- How much redundancy to build in
- What level of availability to accept

It influences these judgments.

In other words, cost is **both a result of design and a constraint on design.**

6. What This Chapter Is Showing

What has been examined in this chapter
is not a discussion of how to reduce cost.

It is the fact that cost directly reflects structure.

- What is not visible cannot be optimized
- What has ambiguous responsibility is not improved
- When structure changes, cost also changes

In the cloud-native world,
cost is not something managed after the fact.

It becomes something included in design from the beginning.

Furthermore, cost not only appears as a result of design —
it is also a factor that constrains which design is chosen.

7. The Question That Remains

When cost is determined by structure:

- At what unit is cost grasped?
- How is responsibility divided?
- How is continuous optimization achieved?

Chapter 4: Redefining Structure in the Age of AI

0. Cloud Native and AI Are in the Same Flow

What has been examined up to this point is the structure of:

- Separation of execution units
- Making control a mechanism
- Abstraction of environment
- Redefinition of responsibility

Cloud native emerged to keep systems viable under the assumptions of change, distribution, and uncertainty.

AI also appears on top of this flow.
It moves judgment that people were taking on to the outside, and makes it an object to be handled as a mechanism.

This is the same as the structure seen in cloud native.

1. AI Is Also a Technology That Changes How Responsibility Is Divided

AI appears to be adding “new functions,” but that is not the essence.
What is changing is that **part of the judgment and processing that people were taking on has moved to the outside**.

Work that was done manually is automated, part of judgment is delegated to the mechanism,
and the range in which people are involved changes.

This is the same as the structural change seen in cloud native.

2. “Those Who Use” and “Those Who Keep Viable”

With the spread of AI, a major divergence has emerged.
Those who use AI to obtain results, and those who use AI to keep mechanisms viable.

This difference is not a difference in skills.

It is a difference in **how far one treats as their own responsibility**.

- Whether to use results as they come out
- Whether to understand how those results were generated
- Or whether to design the mechanism itself

This is where the way value is produced diverges.

3. “Only Using” AI Cannot Remain Viable

With AI, why a particular result occurred, where errors arose,
and in what range it can be used are difficult to see.

If results are continued to be used in this state:

- Errors go unnoticed
- Reproducibility cannot be maintained
- Problems grow

As a result, **even while using AI,**
the system cannot continue to remain viable.

4. Whether It Can Be Observed Becomes the Divergence

What becomes important in this situation is **whether it can be observed**.

- Which input connected to which result
- Where judgment was made
- In what range it can be trusted

Without being able to grasp these, AI cannot be controlled.

As seen in Part 4, **what cannot be observed cannot be controlled.**
AI is no exception.

5. What Remains Unchanged

Even with AI, there are things that do not change.

- How far to hold responsibility
- What to keep viable
- Where to draw the boundary

These do not change even with AI.
Rather, if these remain ambiguous while using AI, problems grow.

Technology changes, roles change, work changes.
Within all of that, what does not change is **the ability to define what to keep viable**.

The structure seen in cloud native connects directly into the age of AI.

- Being able to observe
- Being able to control
- Responsibility being divided

Those who can handle these will continue to produce value in the next era as well.

6. What This Chapter Is Showing

What has been examined in this chapter
is not how to handle a new technology called AI.

It is the fact that the structure seen in cloud native
applies directly to other domains as well.

- Separation of execution units
- Making control a mechanism
- Abstraction of environment
- Redefinition of responsibility

This structure does not change even when technology changes.
AI also rests on top of this structure.

What matters is not the technology itself.

Defining what to keep viable,
and deciding where to place that responsibility

Whether this judgment can be made determines whether value can be produced.

Part 5 Summary

Structure Is Connected to Society

In Part 5, themes such as ways of working, how value is produced, cost, and AI were addressed.
At first glance, they appear to be scattered themes, but in reality all of them rest on the same structure.

What was examined in Parts 1 through 4

- Separating execution units
- Delegating control to the mechanism
- Abstracting the environment

All of this is a decomposition of responsibility — of what to take on, and how far.

In Part 5, how that division of responsibility influences ways of working, value, cost, and AI was examined.
Even when handling the same technology, the value provided is entirely different depending on how far one treats as within their own responsibility.

Cost does not arise after the fact — structure is directly reflected in it.
AI is also a technology that changes how responsibility is divided, and is an extension of structure.

Technology and means continue to change. But there are things that do not change.

- What to keep viable,
- How far to hold responsibility,
- Where to draw the boundary.

These do not change in any era.

What this textbook has addressed is not tools or procedures.

It is structure.

- How to divide execution units
- Where to place control
- How to divide responsibility
- How to observe state

By understanding these, a foundation is created that can be applied even when technology changes.

Cloud native is a discussion of technology,
but what lies beyond it is **the ability to handle structure**.

In a changing environment, taking uncertainty as an assumption, keeping a distributed system viable.
This ability connects not only to technology, but directly to business and society as well.

The Question That Still Remains

Having examined structure up to this point:

- How far does one take on?
- What does one keep viable?
- At which boundary does one stand?

How is that judgment made?

Part 6: Conclusion

1. On Handling Structure

What this textbook has addressed is not cloud native as a technology.

It is the structure of how to keep a system viable
under the assumptions of change, distribution, and uncertainty.

■ Part 1 examined the assumptions.

The environment continues to change, systems are distributed, and uncertainty is unavoidable.
Under these assumptions, the traditional approach cannot remain viable.

■ Part 2 experienced this firsthand.

Units divided, connections formed,
state drifted, communication failed.

A system always contains elements of instability.

■ Part 3 examined how to handle that state.

- How to divide
- How to deliver
- How to connect
- How to behave

All of these are judgments as design.

■ Part 4 examined how to observe that state.

What matters is not what is visible, but what can be judged.

Observation is not the act of grasping state.
It is the act of obtaining information that makes judgment possible.

■ Part 5 examined how this structure connects to society.

- Ways of working
- How value is produced
- Cost
- AI

All of these are determined
by the structure of how far responsibility is held.

What has been examined throughout is consistent.

- Technology changes
- Methods also change
- Roles also change

But there are things that do not change.

- What to keep viable.
- How far to hold responsibility.
- Where to draw the boundary.

Understanding structure means
being able to define these for oneself.
When one becomes able to handle structure, technology no longer takes control.

Even when new tools appear, even when new methods emerge,
one can judge what they resolve and what they take on.

When one becomes able to handle structure, the way problems appear changes.
Rather than surface-level events,
the arrangement of responsibility behind them and the drift in assumptions become visible.

When one becomes able to handle structure, the way value is produced changes.
Rather than completing tasks, one comes to hold responsibility for keeping things viable.

The purpose of this textbook is not to increase knowledge.
It is to create a state in which judgments can be made.

2. The Question That Still Remains

Having examined structure up to this point:

- What does one keep viable?
- How far does one take on responsibility?
- Which structure does one choose?

How is that judgment made?

This question is not something anyone prepares for you.

It is something that continues to change with the situation,
and something one must continue to face for oneself.

The structure examined throughout this textbook exists for that purpose.

Then, ask yourself again.

- **What do you keep viable?**
- **How far do you take on responsibility?**
- **How do you choose that structure?**

Appendix: Bridge to Practice — Certifications and Learning Roadmap

The Role of This Appendix

Throughout this textbook, cloud native has been organized not as a collection of technologies, but as a structure: **how to observe, control, and operate a distributed environment that operates under the assumption of change.**

In actual learning and professional practice, however, this knowledge is organized through certification frameworks and technology categories.
The purpose of this appendix is not exam preparation itself.

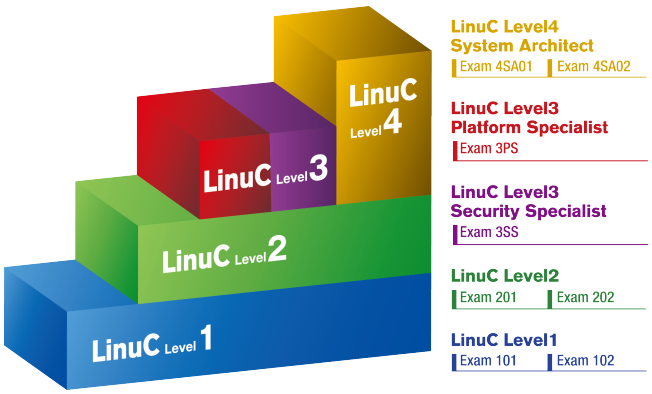
Its purpose is to provide a bridge to the next stage of learning by clarifying:

- which knowledge frameworks the structural understanding built in this textbook connects to,
- what kinds of roles and responsibilities that understanding leads toward,
- and what to learn next to deepen that understanding.

1. Introduction to the Linux Engineer Certification "LinuC"

The Linux Engineer Certification "LinuC" is a technical certification that proves the skills necessary for everything from system construction to operation and management required of IT engineers in the Cloud/DX (Digital Transformation) era. It broadly covers technical domains from architectural design to system construction and operation management. Through the acquisition of four certification levels, you can step-by-step and reliably acquire and prove the skills that are in demand.

The formulation of LinuC's exam topics and exam development is carried out by a community of high-level IT engineers who are active in the field. Therefore, it covers technical areas used as industry standards globally, and the content tests the knowledge and practical skills truly needed at the front lines of system development and operation management. As a result, unlike traditional technical certifications that remain confined to legacy Linux domains, it has become a certification that is sufficiently useful for IT engineers aiming to be active both domestically and internationally.



LinuC Certification Roadmap

LinuC Level 1

Proof of a "job-ready" engineer who understands computer systems and can perform basic operations and system management of Linux systems, including virtual environments (ITSS Level 1).

LinuC Level 2

Proof of an engineer who can perform design, introduction, maintenance, and problem-solving based on architecture for Linux system design and network construction, including virtual environments (ITSS Level 2).

LinuC Level 3 Platform Specialist and Security Specialist

Proof of a specialist who can build and operate Linux platforms with flexibility and high availability through technologies such as virtualization and automation, and who can also implement layer-by-layer hardening from OS to middleware, build authentication and authorization infrastructure, and execute countermeasures against attacks for secure Linux systems (ITSS Level 3).

LinuC Level 4 System Architect

Proof of an advanced engineer who can oversee the entire system lifecycle, including on-premises/cloud and physical/virtualized environments, and design and build the optimal architecture (ITSS Level 4).

For more details about LinuC, please refer to the following website:

```
https://linuc.org/en/
```

https://linuc.org/en/

2. KCNA (Kubernetes and Cloud Native Associate)

What Is KCNA

KCNA is a certification offered by the CNCF (Cloud Native Computing Foundation).
What this certification confirms is not Kubernetes operational skill itself.

- What kind of thinking does cloud native represent?
- Why does a distributed structure become necessary?
- Why do declarative management and Observability matter?

It is a certification that examines structural understanding of cloud native as a whole.
In that sense, it connects naturally to the knowledge framework developed throughout this textbook.

Exam Overview

Item	Details
Format	Multiple choice
Duration	90 minutes
Questions	60
Passing threshold	Not disclosed (approximately 75%)
Delivery	Online
Provided by	CNCF / The Linux Foundation

※ Exam content and format are subject to change. Please verify official information at the time of registration.

Domains Covered by KCNA

KCNA organizes the cloud native world into the following broad domains.

1: Cloud Native Fundamentals (Where everything begins.)

This domain covers the foundational concepts of cloud native:
containers, microservices, declarative management, and related thinking.

> "**Why did containers become necessary?**"

2: Kubernetes and Container Orchestration (Making distribution manageable.)

This domain covers the basic building blocks — Pod, Node, Control Plane, Service —
and the management structure of distributed environments.

> "**Why did a system for managing distribution emerge?**"

3: Cloud Native Architecture (Designing with change as the baseline assumption.)

This domain covers microservice design, APIs, stateless structures,
and the design thinking that takes change and scalability as given.

> "**Why does a fixed design eventually reach its limits?**"

4: Observability (The technique of seeing what cannot be seen.)

This domain covers how to understand the internal state of a distributed environment through Monitoring, Logging, and Tracing.

> "**Why does the interior become less visible as a system becomes more distributed?**"

5: Networking (Abstracting the connections.)

This domain covers the mechanisms of connection and communication in distributed environments: Service, DNS, Ingress, and related components.

> "**Why are connection targets not fixed?**"

The relationship between this textbook and these domains is organized in the table below.

Correspondence with This Textbook

Textbook Section	Corresponding KCNA Domain
Prologue / Part 1: Foundational structure of containers and Kubernetes	Cloud Native Fundamentals / Kubernetes Fundamentals
Part 2: Kubernetes in practice and basic operations	Kubernetes Fundamentals / Orchestration
Part 3: GitOps, Service Mesh, distributed design	Cloud Native Architecture / Application Delivery
Part 4: Observability and operational judgment	Observability
Part 5: Continuous change, value, and responsibility	Cloud Native Concepts (related domains)

What matters is not memorizing KCNA terminology.

It is connecting names from a knowledge framework to the structures already understood through this textbook.

3. CKA / CKAD / CKS — Differences from Intermediate and Advanced Certifications, and How to Approach Them

Intermediate Certifications Change the Scope of Responsibility

Where KCNA examines structural understanding of cloud native as a whole, CKA, CKAD, and CKS each address different roles and scopes of responsibility.

What matters is not "**which certification is higher.**"
It is which domain to engage with deeply, and which responsibility to take on.

That distinction shapes the direction of learning.

CKA (Certified Kubernetes Administrator)

■ Keeping a distributed environment running without stopping.

CKA is a certification centered on the operation and management of Kubernetes clusters.

Through node management, cluster configuration, network settings, failure response, and scaling, it addresses "**keeping a distributed environment stably maintained.**"

More than building a system,
this certification demands an operational practice of grasping state
and responding to change within an environment where failures and changes continue to occur.

> "**Why does operation itself become design?**"

CKAD (Certified Kubernetes Application Developer)

■ Building with continuous change as the baseline assumption.

CKAD is a certification covering the design and deployment of applications that run on Kubernetes.

Through Deployment, ConfigMap, Secret, Service, Helm, CI/CD, and related tools,
it addresses "**application design with change as the baseline assumption.**"

Rather than making an application self-contained in isolation,
the focus is on how to connect it within a distributed environment
and how to continuously apply changes to it.

This domain concerns application design for the cloud native era.

> "**Why do applications also become distributed by assumption?**"

CKS (Certified Kubernetes Security Specialist)

■ Maintaining safety while permitting change.

CKS is a certification that specializes in security within Kubernetes environments.

Through RBAC, NetworkPolicy, Supply Chain Security, Runtime Security, and related topics,
it addresses how far to permit, where to apply control, and how to maintain safety.

In cloud native environments, as systems become more distributed, connection points and Responsibility Boundaries also multiply.

Furthermore, Kubernetes is not a self-contained system.

Systems are kept viable through the combination of multiple technologies and OSS components:
applications, containers, operating systems, networks, and more.

For that reason, CKS requires not only understanding of Kubernetes configuration
but also understanding of the surrounding technologies and infrastructure layers.

Security does not become viable through any single component alone.
It must be considered while maintaining awareness of boundaries and Responsibility Boundaries across the entire system.

Within that, how to design boundaries,
how to control risk, and how to maintain safety while preserving flexibility —
these questions form the center of this domain.

CKS is both a Kubernetes security certification
and a certification for thinking about how to protect an entire cloud native environment.

> "**Why do Responsibility Boundaries become more critical as systems become more distributed?**"

[!WARNING]
CKA certification is required to register for CKS.

Where to Begin

There is no single correct order for learning.

What matters is "**which domain one wants to deepen.**"

Interest	Next Option
Organize the overall cloud native landscape	KCNA
Deepen Kubernetes operations	CKA
Deepen application design	CKAD
Deepen security and boundary control	CKS

※ KCNA provides a starting point for those who want to organize the structural understanding developed in this textbook within a knowledge framework.

Certifications are not a destination.
They are one perspective for deepening understanding.

Building on the structural understanding developed in this textbook,
what matters is thinking about which responsibilities and roles one wants to develop further,
and connecting to the next stage of learning.

4. Example Learning Plans

■ There Is No Single Correct Path

In cloud native learning, almost nothing is completed through a single technology alone.

- Operations
- Development
- Design
- Security
- Observation

Each connects to the others
to constitute the system as a whole.

For that reason, what matters is not only "**where to start,**"
but "**which understanding to deepen.**"

The way learning progresses is also not singular.

There are approaches that expand outward from the whole picture step by step,
and approaches that go deeper from a specific role or responsibility.

The following presents representative learning progressions as examples.

1: For Those Who Want to Learn Progressively

▼ First understand the whole, then go deeper.

A learning plan that grasps the overall flow of cloud native
while gradually moving into practice and specialized domains.

Phase	Learning Content
Step 1	Linux / Docker fundamentals
Step 2	Structural understanding via this textbook
Step 3	Organize the knowledge framework via KCNA
Step 4	Kubernetes hands-on
Step 5	Branch into CKA / CKAD / CKS

"**Why does partial understanding alone make overall operations more difficult?**"

2: For Those Who Want to Move Toward Operations

▼ Understand in order not to stop.

A learning plan oriented toward Kubernetes operations, failure response,
and deepening the practice of stable operation.

Phase	Learning Content
Step 1	This textbook, Parts 1 - 4
Step 2	Kubernetes basic operations
Step 3	Monitoring / Logging
Step 4	CKA
Step 5	Real operations / failure response

Within an environment where failures and changes continue to occur, how is stability maintained?
This path goes deeper into operational judgment and state awareness in distributed environments.

"**Why does 'operations' itself become critical in a distributed environment?**"

3: For Those Who Want to Move Toward Development and Design

▼ Build with the assumption of continuous change.

A learning plan oriented toward application design, continuous change,
and design adapted to distributed environments.

Phase	Learning Content
Step 1	This textbook, Parts 1 - 3
Step 2	Kubernetes basic operations
Step 3	GitOps / CI/CD
Step 4	CKAD
Step 5	Microservice design

Rather than making an application self-contained,
this path develops understanding of how to connect
and how to operate within an environment of continuous change.

"**Why must design also take 'change' as its baseline assumption?**"

4: For Those Who Want to Move Toward Security

▼ Protect while permitting change.

A learning plan oriented toward boundary design
and Responsibility Boundaries in cloud native environments.

Phase	Learning Content
Step 1	This textbook, Parts 4 - 5
Step 2	Kubernetes fundamentals
Step 3	RBAC / Policy management
Step 4	CKS
Step 5	Supply Chain Security

In cloud native environments, as systems become more distributed, connection points and Responsibility Boundaries also multiply.

Within that, how far to permit and where to apply control —
this path develops the understanding of how to design boundaries
while preserving flexibility.

"**Why do 'boundaries' become more critical as systems become more distributed?**"

■ In Closing

Cloud native is not a world that can be completed through a single technology.

What matters is not only increasing knowledge,
but expanding "**from which Observation Points one sees the system.**"

There is no single correct answer to a learning plan, either.

Cloud native itself is a world that takes change as its baseline assumption —
and so the way of learning is likewise not something that becomes fixed.

If this textbook can serve as an entry point into that world, that is its purpose.

Summary

Throughout this textbook, cloud native has been organized not as a set of technical terms or products, but as a structure: "how to observe, control, and operate a distributed environment that operates under the assumption of change."

Technologies such as containers, Kubernetes, GitOps, Service Mesh, and Observability
are treated not as things to be memorized individually,
but to understand "**why that structure became necessary.**"

In the world of cloud native, systems continue to change at all times.

Configuration is not fixed, roles continue to shift,
and the responsibilities and perspectives required keep expanding.
For that reason, what matters is not “memorizing the correct answer,”
but “**how to grasp structure within change.**”

Certifications and learning roadmaps are simply one entry point for that purpose.

The certification frameworks of KCNA, CKA, CKAD, and CKS
are, at their core, not only for increasing knowledge —
they can also be understood as a way to clarify “**from which perspective one sees the system.**”

There is no single correct path for how to learn.

Some will deepen operations; others will deepen design.
Some will move toward security or Observability.

What matters is to continue asking: what perspective does one want to use when viewing systems,
and what responsibilities does one want to take on.

If this textbook can serve as an entry point
for grasping this ever-changing world of cloud native as structure,
that is its purpose.

Cloud Native Textbook

1.0.0

Publisher LPI-Japan