



Enabling safe & secure modular UPdates, UPgrades and DynAmic Task reallocation and Execution for Software Defined Vehicles

D2.2. Container Prototype [initial] V1.0

Kernkonzept (KK)

30.06.2026



Funded by
the European Union

The project, its research and all other activities leading to these results have received funding from the European Union and the granting authority CINEA under Grant Agreement No 101203060

PUBLIC

© Copyright 2025 UP2DATE4SDV Consortium Partners
101203060 — UP2DATE4SDV — HORIZON-CL5-2024-D6-01

Disclaimer:



Funded by
the European Union

Funded by the European Union. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the granting authority (European Climate, Infrastructure and Environment Executive Agency - CINEA). Neither the European Union nor the granting authority can be held responsible for them.

All rights reserved.

Deliverable Information		
Deliverable Number	D2.2	
Deliverable Name	Container Prototype initial design	
Nature	Report (R)	
Dissemination Level	Public	
Contractual Date of Delivery	30.06.2026	
Actual Submission Date	01.07.2026	
Lead Beneficiary	KK	
Lead Author	Marcus Hähnel	
E-Mail	marcus.haehnel@kernkonzept.com	
Reviewing Beneficiaries	KK (WPL)	Marcus Hähnel
	DLR	Henning Schlender Patrick Uven
Work Package(s)	WP2	
Related Task(s)	Task 2.2	Reconfigurable and resource controllable container middleware implementation
Keywords	Containers, L4Re, Microkernel, OCI, Container Runtime	

Project Information	
Contract #	101 203 060
Fixed Project Start Date	01/07/2025
Project Duration	36 months
Model Grant Agreement based on	HE MGA — Multi & Mono - V1.2
Call	HORIZON-CL5-2024-D6-01
Topic	HORIZON-CL5-2024-D6-01-01
Type of Action	HORIZON-RIA
Service	CINEA/C/03
Project Website	https://up2date4sdv.eu/

Document History		
Revision	Date	Description (add and/or delete)
V0.1	17.06.2026	Initial version
V0.2		Updated version for partners
V0.3	25.06.2026	Version for review
V1.0	30.06.2026	Final and reviewed Version

Table of contents

Table of contents.....	5
Executive Summary	7
Acronyms and Abbreviations	8
List of Figures.....	9
List of Tables.....	9
References and Background Material	9
1. Introduction	11
1.1. Project Outline	11
1.2. Technical Deliverable-Map	12
1.3. Work-Package Outline	13
1.3.1. Work-Package Objectives	13
1.3.2. Work-Package Description	14
1.3.3. Work-Package Status.....	14
1.4. Purpose and Scope of the Deliverable.....	14
2. Background and Related Work	16
2.1. Container Technology and the OCI Model.....	16
2.2. Container Technology and the OCI Model.....	16
2.3. Comparable Systems.....	17
2.3.1. Linux Ecosystem and Kata Containers.....	17
2.3.2. QNX Container Architecture.....	17
3. Conceptual Model and Terminology.....	19
3.1. Glossary.....	19
3.2. L4Re Compartments and Containers	19
4. Requirements	21
4.1. Goals	21
4.2. Functional Requirements.....	21
4.3. Non-functional Requirements	22
5. Design of the L4Re Container Service	22
5.1. Architecture Overview	23
5.2. Container Manager	24
5.3. Container Runtime.....	24
5.4. Execution Environment and Resource Mapping.....	25

5.5.	Interaction and User Interfaces	25
5.6.	Virtual Machine Integration (uvmm)	26
6.	Implementation Roadmap	27
7.	Open Questions and Design Decisions.....	28
8.	Summary and Conclusion.....	28
9.	Outlook.....	29

Executive Summary

This deliverable presents the initial design of a container runtime for the L4Re Hypervisor & Operating System Framework, developed within Task 2.2 of Work Package 2. Its purpose is to enable the execution of industry-standard containers on L4Re while preserving the isolation, resource-control and real-time properties that microkernel-based systems provide. The document is aimed at technical reviewers of the project and consortium partners who will build upon, integrate with, or evaluate the container subsystem in later phases. Readers can expect a survey of the relevant container standards and comparable systems, a conceptual model that reconciles the Open Container Initiative (OCI) model with L4Re's capability-based resource model, a proposed set of functional and quality requirements, and an architectural design that separates a *container manager* from a *container runtime* and connects them to existing L4Re components. The design follows the OCI Runtime Specification so that users familiar with mainstream container tooling can operate L4Re containers with concepts they already know, and so that compatibility with ecosystems such as the Kubernetes Container Runtime Interface (CRI) can be added incrementally. The work directly supports the project's OS/Middleware Abstraction Layer (OAL) objective by providing a portable, standards-aligned deployment and execution unit for software-defined-vehicle workloads.

A theme runs through the design: where mainstream container platforms isolate workloads by progressively constraining an otherwise permissive shared kernel, L4Re confines them by construction. Under its capability-based, least-authority architecture a container holds only the resources explicitly granted to its compartment.

The main message is that the OCI container model can be realised on L4Re without sacrificing the platform's isolation guarantees, by mapping container resource requirements onto explicitly granted L4Re resources rather than onto implicit operating-system ambient authority. The conclusion is that an incremental, iteration-driven implementation path, beginning with a minimal manager/runtime pair driven by the L4Re startup scripting environment `ned`, is both feasible and the recommended way forward.

Acronyms and Abbreviations

Abbrev.	Full Name
AD	Autonomous-Driving
API	Application Programming Interface
CA	Consortium Agreement
CAD	Connected and Automated Driving
CCAM	Connected Cooperative and Automated Mobility
CLI	Command-Line Interface
CNI	Container Network Interface
COTS	Commercial Off-The-Shelf
CRI	Container Runtime Interface
CSA	Coordination and Support Action
D	Deliverable
DPDK	Data Plane Development Kit
EC	European Commission
ECU	Electronic Control Unit
EU	European Union
GA	Grant Agreement
HAL	Hardware Abstraction Layer
HW	Hardware
IP	Intellectual Property
IPC	Inter-Process Communication
JU	Joint Undertaking
K8s	Kubernetes
M	Month
MBSE	Model-Based System Engineering
MS	Milestone
OAL	OS/Middleware Abstraction Layer
OCI	Open Container Initiative
ORP	Orchestration and Reconfiguration Plane
PC	Project Coordinator
POLA	Principle of Least Authority
PU	Public
R	Report
SASE	SAfety and SEcurity
SD-CCAM-V	Software-Defined CCAM vehicles
SDV	Software Defined Vehicle
SEN	Sensitive
SOTA	Software-Over-The-Air
SR-IOV	Single-Root I/O Virtualization
SW	Software
ttRPC	Leightweight (Tagged) gRPC variant used by container runtimes

TL	Task Leader
VM	Virtual Machine
VMM	Virtual Machine Monitor
WP	Work Package
WPL	Work Package Leader

List of Figures

Figure 1: Overview of the UP2DATE4SDV work-packages and their Interaction.....	12
Figure 2: Overview of the UP2DATE4SDV technical deliverable map.....	12
Figure 3: Conceptual relationship between an L4Re compartment (resource set / isolation domain) and an OCI container (portable application + dependencies)	20
Figure 4: Architecture of the L4Re Container Service	24

List of Tables

Table 1: List of Terms used in this section	Error! Bookmark not defined.
Table 2: List of functional requirements.....	Error! Bookmark not defined.
Table 3: List of non-functional requirements	Error! Bookmark not defined.
Table 4: Mapping of Container requirements to L4Re mechanisms ..	Error! Bookmark not defined.

References and Background Material

Internal Background

- [1] D2.1 – Requirement Specification
- [2] D3.1 – Requirements & Drafts of the Orchestration and Reconfiguration Plane (ORP)

External Background

- [3] Open Container Initiative: *Runtime Specification* (lifecycle and `config.json`).
<https://github.com/opencontainers/runtime-spec>
- [4] Open Container Initiative: *Image Specification* (image manifest, layers, image index).
<https://github.com/opencontainers/image-spec>
- [5] Open Container Initiative: *Distribution Specification*.
<https://github.com/opencontainers/distribution-spec>
- [6] Kubernetes: *Container Runtime Interface (CRI)*.
<https://kubernetes.io/docs/concepts/architecture/cri/>
- [7] runc: OCI reference runtime. <https://github.com/opencontainers/runc>
- [8] containerd: industry container runtime/daemon. <https://containerd.io/>
- [9] Kata Containers: VM-isolated containers. <https://katacontainers.io/>

- [10] BlackBerry QNX: *QNX Containers* (OCI- and Kubernetes-compliant container runtime on the QNX microkernel). <https://blackberry.qnx.com/en/products/foundation-software/qnx-containers>
- [11] BlackBerry QNX: *QNX OS System Architecture* (SDP 8.0): microkernel, process manager (procnto), resource managers, and message-passing services.
https://www.qnx.com/developers/docs/8.0/com.qnx.doc.neutrino.sys_arch/topic/about.html
- [12] L4Re Operating System Framework. <https://l4re.org/>

1. Introduction

The European research project UP2DATE4SDV (Enabling safe & secure modular UPdates, UPgrades and DynAmic Task reallocation and Execution for Software-Defined Vehicles), aims to develop a comprehensive ecosystem for seamless and efficient software updates, hardware upgrades and situation-dependent reconfigurations of software-defined vehicles.

1.1. Project Outline

The project focuses on the definition and development of three abstraction layers

- Hardware Abstraction Layer (HAL)
- OS/MW Abstraction Layer (OAL) for the operating system (OS) and middleware (MW)
- Exploration and prototyping of a secure orchestration and reconfiguration layer (ORP) between the vehicle and the cloud.

This will improve the reliability and performance of the CCAM system, extend vehicle life and reduce electronic waste. In addition, the project's new modularity concepts will accelerate the transition to safer, more sustainable automated driving and enable incremental steps towards Level 4 automation and longer vehicle lifecycles

The overall UP2DATE4SDV goal is to create a comprehensive ecosystem for updatable, upgradable, and reconfigurable software-defined connected and automated vehicles. This will be achieved by developing essential abstraction layers, advanced orchestrators, and an automated robust realization environment, all while adhering to the highest safety and cybersecurity standards. To achieve this vision, the project will target the following objectives:

- O1. Creation of a reconfigurable cloud connected zonal vehicle architecture blueprint based on heterogeneous COTS HW (exploiting and building on a unified HAL)**
- O2. Establishing a resource composable SW deployment, execution and communication reference layer, on top of existing open-source hypervisor and OS stacks (exploiting and building on a unified OAL)**
- O3. Realization of a dynamic update, orchestration & reconfiguration plane**
- O4. Provide harmonized and simplified development, integration and validation processes and framework for SDV applications**
- O5. Assess joint Safety and Security (SASE) of all elements from O1, O2 and O3**
- O6. Demonstration of UP2DATE4SDV solution on several CCAM relevant Use Cases**
- O7. Active participation around CCAM and SDV Initiative community and ecosystem**

To this goal, the demonstrators and use-cases (developed in WP5) define requirements (pink arrows) for the HW and the HAL (developed in WP1), the SW and the OAL (developed in WP2) as well as for the ORP (developed in WP3). Based on that, WP1, WP2 and WP3 provide a basic functional HW-SW-demonstrator for the use-cases – UC1- after Phase 1 (M6), UC2, UC3, and UC4 after Phase 2 (M24), and finally UC5 and UC6 after Phase 3 (M36).

Besides the pure communication of results (green arrows) the proper integration of the results of WP1, WP2, WP3, and WP4 within the use-cases depends on three further cooperation aspects: the safety and security assessment (yellow arrows), the update-, upgrade-, and reconfiguration-processes and the involved component-interfaces (purple arrows), and the cloud-based MBSE-representation of

actual HW-/SW-components during the DevOps cycle establishing the real-world to cloud connection according to digital twins (blue arrows). Finally, while WP6 oversees the dissemination, communication, and exploitation of the project's results, WP7 coordinates all the WP and their communication, internally as well as externally to the SDV-Initiative and to other CCAM-partnerships. Figure 1 summarizes the described interaction and communication between the work-packages (WPs) of UP2DATE4SDV.

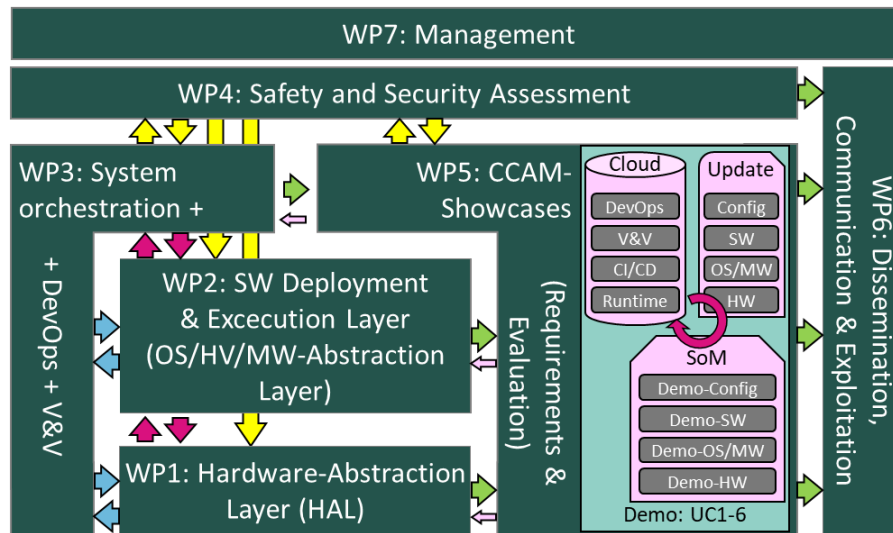


Figure 1: Overview of the UP2DATE4SDV work-packages and their Interaction.

1.2. Technical Deliverable-Map

UP2DATE4SDV has defined 38 deliverables of which 10 have an organizational nature and 28 a technical nature. To provide orientation to the reader, Figure 2 shows a technical deliverable map that gives a structural overview of all technical deliverables and their content-wise relation.

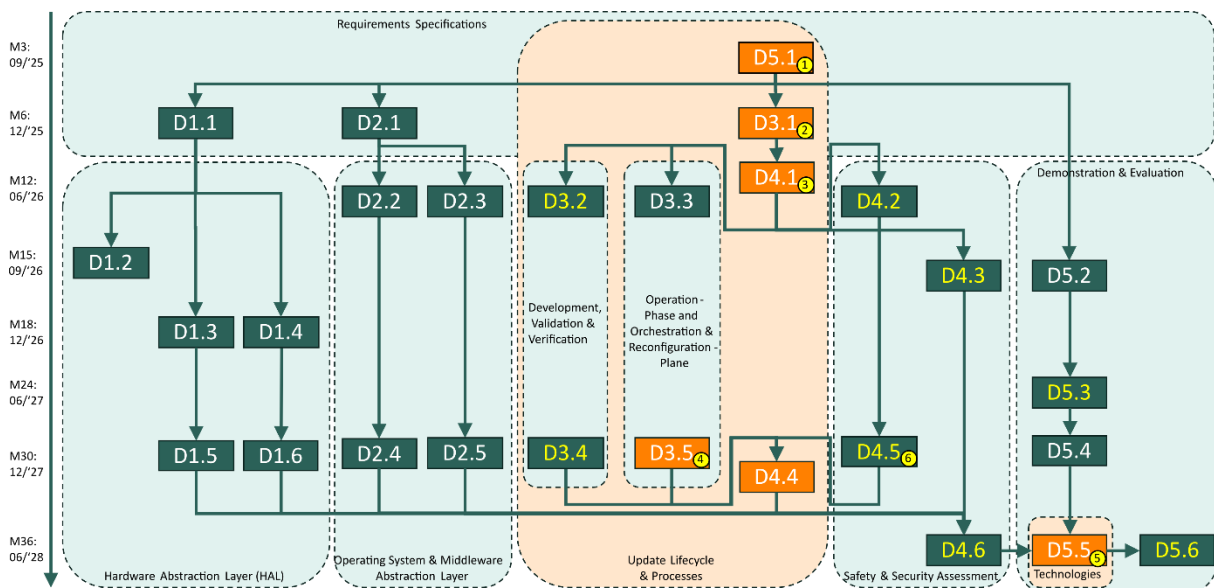


Figure 2: Overview of the UP2DATE4SDV technical deliverable map, showing the deliverables of the technical work-packages WP1-WP5, with restricted deliverables highlighted by the yellow text, a main-track highlighted by the orange nodes, and a recommended reading-order in the yellow circles. The arrows between deliverables are indicating the content development.

Deliverables are shown as rectangles, which are connected by arrows if the content of one deliverable is based on the contents of previous ones. The text in the rectangles shows the deliverable number and it is highlighted in yellow, in case of restricted deliverables, whereas public deliverables are labelled in white. The light-green and light-orange areas form technical content-categories, close but not equal to the work-packages. They show the hardware abstraction layer (HAL) in WP1, the Operating-System and Middleware Abstraction-Layer (OAL) in WP2, the Safety and Security Assessment in WP4, and the Demonstrators and the Evaluation in WP5, but they combine the Orchestration and Reconfiguration-Plane of WP3 with the safety- and security-analysis and corresponding measurements from WP4, forming the area of the Update Lifecycle and Processes, where Development, Validation and Verification, and Orchestration and Deployment of Updates are covered. Furthermore, the four initial deliverables of the work-packages requirements-definitions are also combined in a separate area at the very top. To get an overall impression of the full project, the main story-track of deliverables is highlighted in orange, and a suggested reading-order is indicated by numbers in yellow circles.

1.3. Work-Package Outline

1.3.1. Work-Package Objectives

Work Package 2 realises the project's second objective (O2): to establish a resource-composable software deployment, execution, and communication reference layer on top of existing open-source hypervisor and operating-system stacks. This is called the unified OS/Middleware Abstraction Layer (OAL). The objective is to support platform-independent development, verification and validation, and deployment of application software, together with its seamless integration, scalability, and maintainability, across the diverse platform configurations and architectures of the dynamically changing cloud-edge continuum of CCAM. Unlocking the full potential of updates, upgrades, and dynamic vehicle-to-cloud and cloud-to-vehicle task reallocation requires a unified software-abstraction layer above the hardware abstraction layer that safely isolates application software and decouples the application layer from dependencies on both the hardware components of the platform and their lower-level software stacks. Built on existing open-source hypervisors, operating systems, and middleware components, the OAL provides an interoperable common interface to the application layer on two sides — locally on each vehicle and online in the cloud, where additional software can be executed and accessed. Container technologies are central to this objective because they can transparently support both cloud-based and vehicle-local execution. The container runtime designed in this deliverable is the realisation of that capability on the L4Re platform. As L4Re can be deployed in the cloud and on vehicle ECUs it is well suited to support easy workload migration strategies.

The key result associated with this objective is a reference implementation of a resource-composable software deployment and execution layer for software-defined CCAM vehicles (key result R3), to which this deliverable contributes the initial container-runtime design. Two of the objective's success criteria bear directly on the container runtime and motivate the design that follows:

- a proof-of-concept implementation of a microkernel container runtime, demonstrated in a relevant environment
- tests confirming that applications running inside a container cannot exceed their allocated computational and memory resources

These criteria are reflected in the isolation and resource-control properties pursued in Sections 4 and 5.

1.3.2. Work-Package Description

Work Package 2 addresses the challenges faced by software in a highly dynamic and reconfigurable software-defined-vehicle world, in which dynamic updates of individual parts of the system must be possible while the system as a whole continues to operate safely and securely. The work package is organised into three tasks. In the first task (T2.1), the partners collect and analyse the requirements, abstractions, and interfaces needed to enable an individually updatable, highly flexible software and communication stack for the automotive high-performance compute-platforms of the future. Building on those initial requirements, and partly in parallel, the work package proceeds with the implementation of the container middleware (T2.2) and the communication plane (T2.3). The communication plane is based on named-data networking and is reported in its own deliverable (D2.3). It is therefore outside the scope of this document.

This deliverable concerns Task 2.2 which is the *Reconfigurable and resource-controllable container middleware implementation*. The main part is the contribution of Kernkonzept (KK), which extends the L4Re Hypervisor and Operating System Framework with resource and interface descriptions and proposes a native container runtime format together with the corresponding infrastructure. This task contribution is complemented within the work package by resource-monitoring work (DLR) and by interface-definition input drawn from the partners' respective main work packages, all of which will steer the design of the OAL as the implementation matures. The native L4Re container runtime whose initial design is presented here is the central element of the KK contribution to Task 2.2, which the task will implement and refine over successive project phases.

1.3.3. Work-Package Status

At the time of writing, WP2 is in its initial design and prototyping phase for both currently running sub-tasks. Task 2.2 has consolidated the goals and requirements for container support, surveyed the relevant standards and comparable systems, and produced the architectural design presented here. The implementation of a minimal prototype is the next step. The work-package is on schedule with respect to the planned effort for this phase, with the design of the container subsystem (this deliverable) representing the principal milestone reached. Open issues, risks, and their mitigations are summarised in Section 7. Dissemination-wise, this deliverable is public and contributes to the project's outward-facing demonstration of standards-aligned, microkernel-based container execution. Due to the early phase with no workable implementation present this has not been disseminated in the form of inclusion into the Open Source L4Re release.

1.4. Purpose and Scope of the Deliverable

Within the UP2DATE4SDV ecosystem, the ability to package, deploy, and execute software in a portable and well-isolated manner is a central enabler. This deliverable addresses the needs at the operating-system level by presenting the initial design of container support for the L4Re operating system. The purpose of this deliverable is to define the initial design of a container subsystem for L4Re. Concretely, the deliverable

- i) states the goals and proposes the requirements for container support on L4Re,
- ii) surveys container standards and comparable systems to ground the design in established practice,
- iii) introduces a conceptual model that relates the OCI container model to L4Re's compartment and capability model,

- iv) specifies an architecture that separates a container manager from a container runtime and integrates them with existing L4Re components,
- v) and lays out an iterative implementation roadmap.

The deliverable is target at the project's technical reviewers and consortium partners who will integrate with or evaluate the container subsystem. It is a *design* document: it defines the intended structure and behavior of the subsystem and the rationale behind the design choices, rather than reporting on a completed implementation. Detailed interface specifications, the final requirements set, and evaluation results are out of scope at this stage and will be produced as the implementation matures in subsequent iterations and deliverables. The main message is that OCI-compliant containers can be supported on L4Re without weakening its isolation guarantees. The conclusion is that an incremental implementation, beginning from a minimal prototype driven by the existing init scripting system in the Ned component of L4Re, is the recommended path.

In project terms, this deliverable provides the initial design on which the container-runtime contribution to key result R3 is built. The results are validated in two phases, at milestones MS3 (this design document) and MS6 (the implementation). The MS6 implementation constitutes the container-prototype deliverable of WP2, and is provided for potential further evaluation within the basic technology implementation and evaluation activities of WP5 (deliverable D5.2). The present document is the design input to that validation track.

2. Background and Related Work

This section establishes the technical context for the design. It summarizes the container model and the standards the design adheres to, and reviews comparable systems whose architectures informed the proposed approach.

2.1. Container Technology and the OCI Model

A *container* packages an application together with all of its dependencies and the parameters required to run it, in a form that is independent of the specific system on which it is executed. This independence is the central value proposition of containers: the same artifact can be moved between hosts that provide a compatible execution interface.

The Open Container Initiative (OCI) standardises this model across three specifications:

- i) The *Image Specification* defines how the file-system content of a container is described as a set of layers stacked on top of one another, how a *manifest* binds those layers and a configuration to a specific architecture-and-operating-system combination, and how an *image index* groups multiple manifests for different targets.
- ii) The *Distribution Specification* defines how images are transferred between registries and hosts
- iii) The *Runtime Specification* defines the on-disk *bundle* (the unpacked root file system together with a `config.json` configuration file) and the container *lifecycle*: the set of states a container passes through (creating, created, running, stopped) and the operations that move it between them (create, start, kill, delete, and a query of current state).

The reference implementation of the runtime specification is `runc`, which on Linux realizes the execution environment using kernel facilities such as namespaces and control groups. Higher in the stack, `containerd` manages images and container lifecycles as a long-running service, and Kubernetes (K8s) orchestrates containers across many hosts. Adhering to the OCI specifications is therefore the most direct way to achieve the project goal of interoperability with existing container artifacts and tooling.

2.2. Container Technology and the OCI Model

Two classes of user-facing interface are relevant to this design. The first is a *command-line interface* that follows the OCI lifecycle operations directly. This is the familiar low-level way of operating a single runtime. The second is the Kubernetes *Container Runtime Interface* (CRI), a programmatic interface between an orchestrator and a runtime. CRI is structured into two services:

- i) a *Runtime Service* that manages the lifecycle of sandboxes and containers
- ii) an *Image Service* that manages images

In the CRI model, a *Pod* is an isolation domain, or sandbox, inside which several containers run while sharing the sandbox's resources. The Pod concept is significant for L4Re because, as Section 3.2 explains, it has a natural counterpart in the L4Re compartment.

2.3. Comparable Systems

2.3.1. Linux Ecosystem and Kata Containers

On Linux the kernel is monolithic: every container on a host shares one large kernel, and the boundaries between containers are drawn *within* that shared kernel by mechanisms such as namespaces, control groups, capability dropping, seccomp filtering, and mandatory-access-control modules. Isolation is therefore *subtractive* and a container begins with broad reach into a wide system-call and kernel attack surface, and confinement is achieved by progressively restricting that ambient authority. Because the entire shared kernel remains reachable, a single defect anywhere in that surface can suffice to break out of a container. The recurring history of container-escape vulnerabilities shows that this boundary, while effective against many threats, is never fully closed. Where stronger isolation is required, the Kata Containers project sidesteps the shared-kernel problem by transparently executing each container inside a lightweight virtual machine, using para-virtualised devices and transports such as virtio-fs and virtio-block for storage and ttRPC for the agent channel, together with acceleration and monitoring technologies (for example SR-IOV and DPDK).

L4Re follows the opposite design philosophy. It is built on a microkernel and an object-capability architecture in which a component holds no authority by default and can act only through capabilities that have been explicitly granted to it. Capabilities are unforgeable references conferring specific rights. Isolation is thus the starting point rather than an end-state approached by constraining an otherwise permissive environment; this is a direct expression of the principle of least authority (POLA). A container on L4Re can reach only the resources placed in its compartment: there is no ambient authority to be subtracted from, and the kernel that mediates these grants is small enough to keep the trusted computing base correspondingly small. While the container manager and runtime that construct the execution environment do themselves become part of that trusted computing base it is a deliberately minimal and well-defined base, not an entire shared operating-system kernel. For this reason, L4Re does not need the Kata approach in order to *obtain* isolation. It confines containers natively, in line with the O2 success criterion. The same transparent VM-execution mechanism nevertheless remains useful on L4Re for a different purpose: running a Linux-native container unchanged inside a virtual machine on Uvmm, the L4Re-native virtual machine monitor to gain compatibility with existing container images (see Section 5.6).

2.3.2. QNX Container Architecture

QNX is, like L4Re, a microkernel operating system, and one that is widely deployed in automotive and other safety-related domains. This makes it the most directly relevant of the comparable systems considered here: it demonstrates that OCI-compatible container support can be provided on a microkernel platform, and the way it structures that support is instructive.

On a QNX machine, containers are operated through two interchangeable user interfaces:

- i) an implementation of the Kubernetes Container Runtime Interface (CRI)
- ii) a command-line interface

Both talk to a common *QNX Container Manager*. The manager interprets each container request arriving through an interface, loads the container from a local or remote container store, monitors the container's lifecycle state, and interacts with the runtime to carry out the requested operations. The construction of the container's environment is delegated to a separate *QNX OCI Runtime*, which creates and deletes the container environment within the QNX system including file-system, network,

and CPU/memory isolation. The runtime then launches and monitors the container process, handles its logging, and stops the process and releases its resources on termination

The L4Re design described in Section 5 exhibits a comparable separation between an interface-facing manager and a system-facing runtime owing to the fact that faithful adherence to the OCI specifications pushes any implementation toward distinguishing the management of images and lifecycle from the low-level construction of the execution environment. The L4Re design arrives at that separation independently, as a consequence of the same standard. The QNX design corroborates that this structure is the natural one for an OCI-compliant microkernel system, and that the approach is viable on a microkernel at all. Where the two necessarily diverge is in the lower half of the runtime: the QNX runtime realizes isolation using QNX mechanisms, whereas the L4Re runtime maps each resource onto explicitly granted L4Re capabilities (Section 5.4).

Beneath the container layer, the two operating systems differ in a way that bears directly on the isolation argument of this deliverable. While both QNX and L4Re are microkernel systems QNX pairs its microkernel with a mandatory process-manager module that provides the POSIX process model and a single global pathname namespace, into which user-space resource managers mount the services they own. Access to those services is then governed by user and group identities, fine-grained process-manager *abilities* (which can be locked to reduce attack surface even for privileged processes), and a type-based mandatory-access-control policy. L4Re is more reductive: its kernel provides only address spaces, threads, scheduling, IPC, and object capabilities, while the process model, naming, and POSIX compatibility are assembled from optional user-space components, with no ambient global namespace. The consequential distinction for containers is one of *designation*. In QNX a resource is named in a globally visible pathname space and access to it is then gated by policy; in L4Re an object capability is at once the name and the authority, so a component can reach only what has been placed in its capability space and there is nothing ambient to be pared back. This is the additive-versus-subtractive distinction made in Section 2.3.1 for Linux applied among microkernels: under the capability model, least authority is the default consequence of how resources are designated rather than a posture configured on top of an otherwise reachable system — which is what bounds the attack surface a container can present and keeps the trusted computing base of its execution environment small and explicit.

3. Conceptual Model and Terminology

3.1. Glossary

The terms in **Error! Reference source not found.** are used throughout this deliverable with the meanings given. They follow OCI usage where applicable.

Table 1: List of Terms used in this section

Term	Description
Bundle	The combination of an image's file-system content and a runtime specification.
Image / Layers	A set of file-system contents layered on top of one another to form the file system a container sees.
Spec	The <code>config.json</code> file as defined by the OCI Runtime Specification.
Manifest	A description binding configuration and image layers for a specific architecture-and-OS combination.
Image Index	A description grouping multiple manifests for differing architecture-and-OS combinations.
Container	The concrete, runnable instance derived from a bundle (and, by extension, the description of its content and dependencies).
Process	The binary executed when a container is run.
Runtime	The entity that sets up the environment required to run a container's process.
Lifecycle	The states a container moves through, as defined by the OCI Runtime Specification.
Status	A specific subset of lifecycle states: creating, created, running, stopped.

3.2. L4Re Compartments and Containers

L4Re's resource model for an isolated part of a system is built on a *compartment*: a defined set of resources made available to the applications executing within it, with the kernel ensuring that those applications can access nothing outside that set. The compartment therefore defines part of the execution environment of a container, and is conceptually similar to the Kubernetes Pod. Both are isolation domains describing the resources available to the workloads inside them.

A container, by contrast, describes an application together with all its dependencies and parameters, and is by design independent of the system on which it runs. These two notions interact at runtime rather than at definition time. When a container is *created*, the compartment it will run in is deliberately not part of the container's description, which keeps the container portable. However, the compartment that hosts the container at runtime must provide resources sufficient to actually run it. Bridging this gap is an administrative responsibility: the administrator must define the host compartment so that its resources meet the container's runtime needs.

This separation of concerns with a portable, host-independent container on one side, and an explicitly provisioned L4Re compartment on the other, is the conceptual foundation of the design (Figure 3). It also distinguishes L4Re's approach from systems where the execution environment is provisioned implicitly by the host operating system: on L4Re every resource the container uses is an explicitly granted capability.

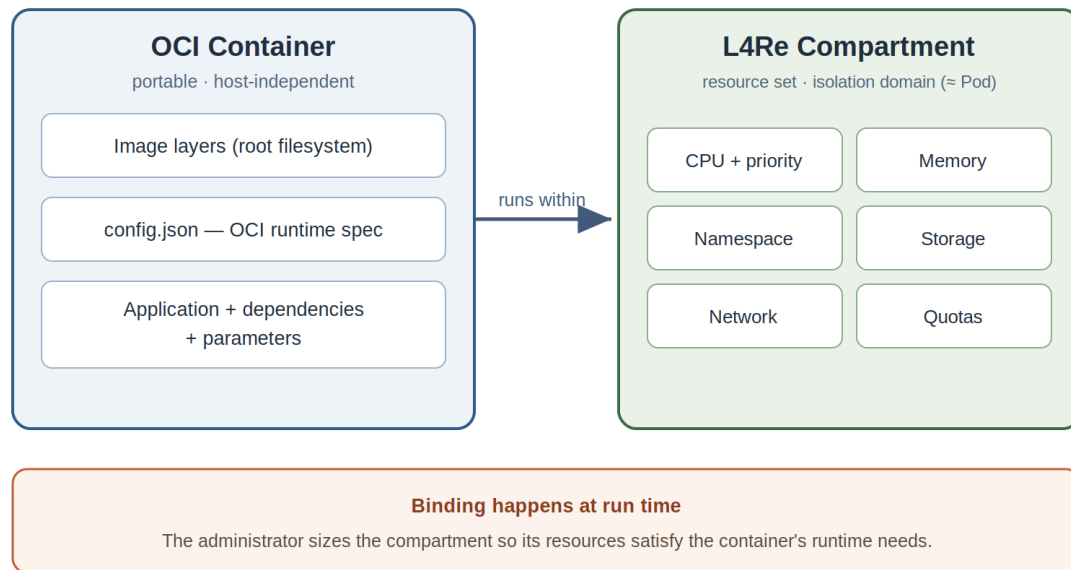


Figure 3: Conceptual relationship between an L4Re compartment (resource set / isolation domain) and an OCI container (portable application + dependencies), showing that the binding happens at run time and that the administrator sizes the compartment to the container

4. Requirements

This section collects the initial requirements for L4Re container support. The requirements are presented as a working set and are expected to evolve as the implementation matures. The requirement identifiers are provisional and expected to change.

4.1. Goals

The design is driven by the following high-level goals for container support on L4Re:

- L4Re shall be able to run entities called *containers* on a booted system.
- A container's contents shall run in an environment isolated from other participants in the system.
- The container lifecycle and user interface shall follow industry standards, so that users new to L4Re can interact with a concept they already know.
- The container support shall support implementation of different kinds of interfaces to manipulate containers to be added. This enables interoperability with existing industry technologies.
- The runtime performance of a container shall be on par with the performance of the same content executed as a native L4Re application.

4.2. Functional Requirements

Table 2: List of functional requirements

ID	Requirement	Description
F.1	Container execution	The system shall instantiate and execute containers on a running L4Re system.
F.2	Isolation	Container contents shall execute in an environment isolated from other system participants, covering file-system, network, and CPU/memory isolation.
F.3	OCI lifecycle	The container lifecycle shall conform to the OCI Runtime Specification: • States: creating, created, running, stopped • operations: create, start, kill/stop, delete, query state
F.4	OCI configuration	The runtime shall construct the execution environment from an OCI config.json and the referenced image content.
F.5	Local CLI	A local command-line interface adhering to the OCI lifecycle operations shall be provided.
F.6	Ned API	A programmatic interface in Lua, usable from Ned configuration scripts, shall be provided.
F.7	Pluggable interfaces	The architecture shall permit additional user interfaces to be added, including a remote interface compatible with the Kubernetes CRI.
F.8	Image loading	The container manager shall obtain image content and specification from the <code>rom/</code> namespace initially, and from local storage in later iterations.
F.9	Lifecycle monitoring	The container manager shall monitor and report container lifecycle state.
F.10	Resource mapping	The runtime shall map the resources requested by a container onto L4Re mechanisms (scheduling, object creation, logging, naming/file-system, storage, network).

F.11	VM-type containers	The system shall interpret OCI VM-type configurations and launch the container within a uvmm-managed virtual machine, amended with L4Re specifics as required.
F.12	Logging	The runtime shall route container process output to the L4Re logging facility under a configurable name.
F.13	Stop control	The manager interface shall allow a running container to be stopped (expected in later iterations).
F.14	Layer integrity	Image layers referenced from local storage shall carry a content hash for integrity verification (expected in later iterations).

4.3. Non-functional Requirements

Table 3: List of non-functional requirements

ID	Requirement	Description
Q.1	Performance	A containerised workload shall achieve runtime performance on par with the same workload executed as a native L4Re application.
Q.2	Confined authority	A container shall be able to access only resources that have been explicitly granted to its compartment. (This realises the Objective O2 success criterion that applications inside a container cannot exceed their allocated computational and memory resources.)
Q.3	Familiarity	The user-facing lifecycle and CLI shall match industry conventions such that users familiar with OCI tooling can operate L4Re containers without L4Re-specific knowledge.
Q.4	Interoperability	The design shall conform to the OCI Runtime and Image specifications to maximise compatibility with existing container artifacts and tooling.
Q.5	Modularity	The separation between interface, manager, and runtime shall allow any one of them to be replaced or extended without changes to the others, via stable and well-defined IPC interfaces.
Q.6	Configurability	Compartment definitions and resource quotas shall be declaratively configurable through Ned scripts.
Q.7	Portability	A container shall remain independent of the specific host instance and shall run on any L4Re system that provides the required host interface.
Q.8	Standards conformance	The host interface required by a container, and its versioning, shall be explicitly defined (see Section 7).

5. Design of the L4Re Container Service

The design is shaped by the guiding principle, that a container's isolation should be a property it holds by construction, not one approximated by constraining a permissive environment. Whereas Linux-based platforms draw container boundaries inside a large shared kernel and reduce ambient authority through namespaces, control groups, and related mechanisms (Section 2.3.1), L4Re grants authority explicitly. Every resource a container uses is a capability placed in its compartment, and nothing beyond those capabilities explicitly placed there is reachable. The architecture described below follows directly from this principle: the runtime's role is precisely to translate a container's declared requirements into an explicit set of L4Re resources, and the trusted computing base that performs this translation is kept small and well-defined.

This principle has a direct impact on performance. Because isolation on L4Re is structural rather than obtained by wrapping each container in a virtual machine, the native execution path incurs none of the per-container virtualization overhead that VM-based isolation approaches such as Kata Containers accept on Linux in exchange for stronger separation. Strong isolation and native-level performance are therefore not in tension on L4Re, which is what makes the native-parity performance goal (requirement Q.1) attainable rather than merely aspirational.

5.1. Architecture Overview

The design separates an interface-facing manager from a system-facing runtime and maps both onto L4Re's component model. As explained in Section 2.3.2, this separation is a direct result from faithful OCI compliance induces. The same standard leads comparable systems such as QNX to a similar split. The three involved roles are:

- **Ned** acts, initially, as the orchestrating entity. It starts the other components, wires them together via IPC, and issues container requests.
- The **Container Manager** is the interface-facing component. It interprets a container request, obtains the bundle (image content plus `config.json`), passes them to the runtime, and monitors lifecycle state.
- The **Container Runtime** is the system-facing component. It implements the OCI lifecycle and constructs the container's execution environment from the `config.json` and image content.

Each container ultimately executes inside an L4Re compartment whose resources have been provisioned to meet the needs of that container. Because the components communicate only over well-defined IPC channels, additional interfaces (such as a CRI front-end) can be attached to the manager without altering the runtime, satisfying the modularity goal (Figure 4).

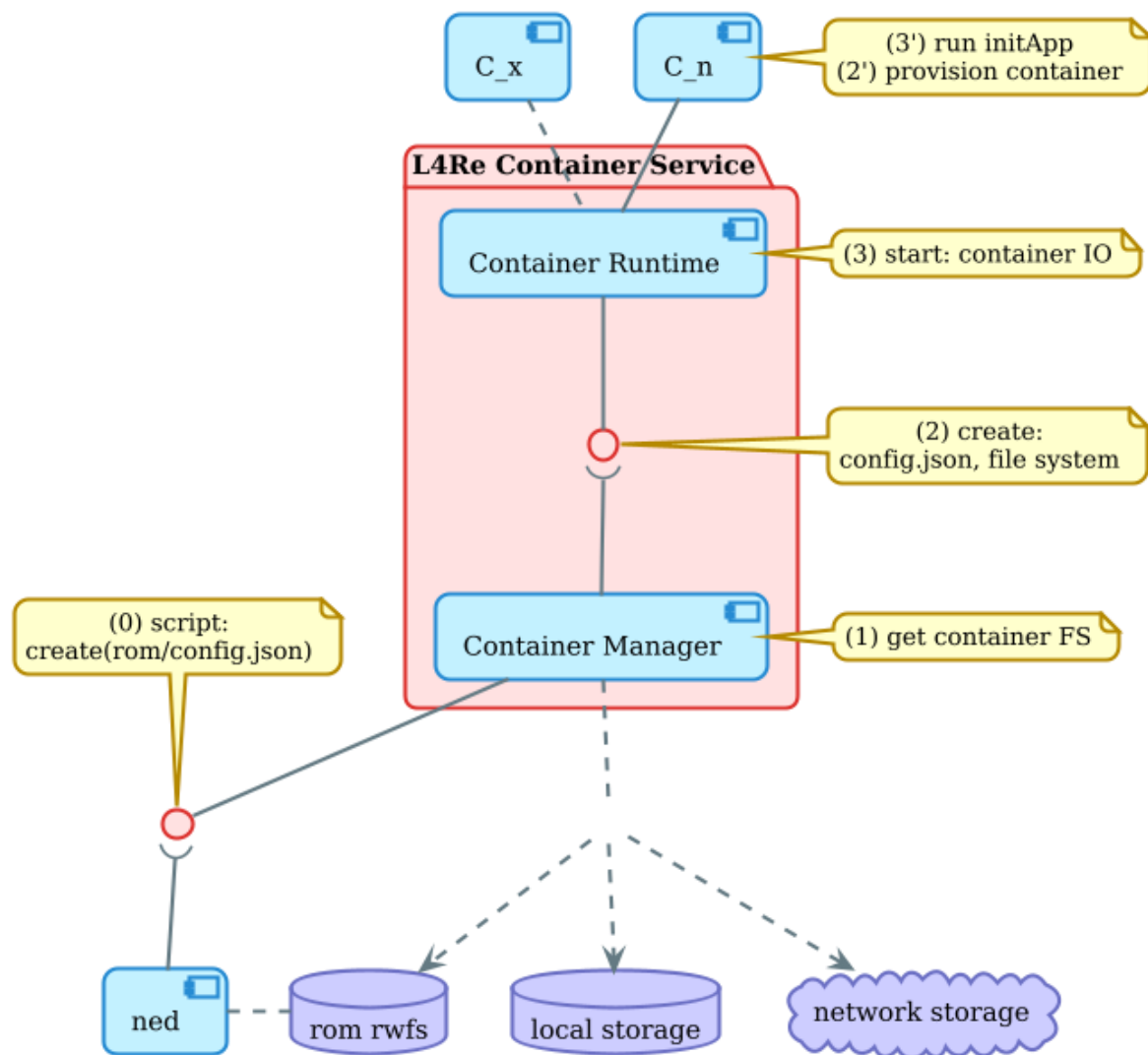


Figure 4: Architecture of the L4Re Container Service. Ned starts and connects the Container Manager and Container Runtime over an IPC channel and issues a config.json. The manager loads the bundle from the rom/rwfs namespace (and later local storage); the runtime

5.2. Container Manager

The Container Manager interprets incoming container requests, regardless of the interface they arrive through. For a given request it locates and loads the corresponding bundle. In the first iteration this loading will happen from the read-only ROM namespace, part of an L4 boot image. In later iterations from local storage. The Container Manager then forwards the `config.json` together with the file-system content to the runtime. Throughout the container's life the manager monitors the lifecycle state of the container such that this state can be reported back through whichever interface is in use. In later iterations the manager's interface is extended so that the orchestrator (Ned, or a remote client) can request that a running container be stopped.

5.3. Container Runtime

The Container Runtime implements the OCI container lifecycle. Given a `config.json` and the associated file-system content, it creates the execution environment, launches the container's process, and supports the remaining lifecycle operations (start, stop/kill, delete). In the minimal

prototype the runtime implements only create and start; the full lifecycle will be implemented in a subsequent iteration. The runtime is the single component that translates abstract container resource requirements into concrete L4Re resources, as described next.

5.4. Execution Environment and Resource Mapping

A container requires access to resources. Where Linux provides these through kernel boundaries that are present by default, L4Re grants them explicitly. The runtime therefore constructs the execution environment by mapping each resource category named in the configuration onto an L4Re mechanism.

Error! Reference source not found. summarises this mapping.

Table 4: Mapping of Container requirements to L4Re mechanisms

Resource	Container requirement	L4Re mechanism
CPU / scheduling	CPU allocation and priority	Scheduling proxy, parameterised with a CPU set and a priority range.
Object creation	Allocation of kernel and user objects	Factory, governed by two quotas
Logging	A named output stream	A Cons (L4Re console multiplexer) instance, which requires at least a name for the container's log.
Naming / file system	Access to the root file system and mounts	An L4Re::Namespace interface (romfs/rwfs/...). An open design question is whether the container can consume an L4Re::Namespace interface directly.
Storage	Backing store for the root file system	The block stack provides a partition. Alternatively a pre-existing file system resource into which the root file system can be unpacked.
Network	Connectivity	Either a layer-2 (Ethernet) or a layer-3 (IP) interface, depending on what the container expects. This is typically realised through a virtio-net device.
Memory	Memory budget	Provisioned within the hosting compartment's quotas.

Two design questions follow directly from the mapping and are expected to be resolved during implementation:

- Can a container image can be expected to interact with an L4Re::Namespace interface for its file system or is native filesystem support a strong prerequisite.
- Do containers expect an Ethernet (layer-2) or IP (layer-3) network interface

Both aspects affect how transparently unmodified container images can be supported.

5.5. Interaction and User Interfaces

The subsystem is designed to be operated through several interfaces by several user roles — including a (possibly remote) administrator and a system integrator. Two categories are foreseen:

Local interfaces:

A command-line interface adheres to the OCI lifecycle and offers the operations:

- Create a container instance from an existing container definition
- Start the created instance
- Stop the running instance
- Destroy the instance, freeing its resources
- Allocate the resources for a container definition's execution environment

In addition, a programmatic API allows containers to be driven from ned scripts.

Remote interfaces:

A remote interface compatible with the Kubernetes CRI is foreseen, attaching to the Container Manager alongside the local interfaces. Because all interfaces share the same manager, support for remote orchestration can be added without changes to the runtime.

5.6. Virtual Machine Integration (uvmm)

The OCI Runtime Specification also defines a VM-type configuration for running a container inside a virtual machine. The design intends to interpret such VM-type configurations and start a uvmm instance accordingly, amended with L4Re specifics where necessary. This serves two purposes: First, when a user explicitly supplies a VM-type configuration, honouring it runs exactly what the user intends. Second, the same transparent VM-execution mechanism that the Kata Containers project uses for isolation can be reused here for an entirely different goal: compatibility. A container image built for Linux expects the Linux kernel interface, which L4Re does not natively provide. Running such an image unchanged inside a Linux VM on uvmm supplies that interface, allowing the existing ecosystem of Linux OCI containers to run alongside L4Re-native containers. The purpose of using a virtual machine here therefore is not the additional *isolation*, intended in the Linux use-case, but a compatibility with the *Linux kernel interface* for pre-existing non-L4Re containers. This is the main difference from Kata, where on Linux the VM compensates for weak shared-kernel separation, a problem that does not arise on L4Re. However, the same concept is well suited on L4Re as a compatibility feature, not a security mechanism.

Both of these VM paths run the container inside a virtual machine and therefore carry the performance overhead inherent in virtualization, which the native execution path (Section 5.1) avoids. The significance is that on L4Re this overhead is incurred only where it is either explicitly desired (honouring an explicit VM configuration), or is needed for compatibility with an unmodified Linux container. It is never the price of isolation, as it is in the Kata model. The native path remains the default and preferred mode of execution, with the VM paths reserved for the cases that necessarily require them.

6. Implementation Roadmap

The subsystem is delivered iteratively. Each iteration produces a working system and yields learnings that shape the next.

Milestone MS3 — Concept of an L4Re container service: The overall concept, captured in the design of Sections 3–5 and illustrated in Figure 3, is established (this deliverable)

Iteration 1 — Minimal implementation: The application `ned` starts the Container Manager and the Container Runtime and connects them over an IPC channel. `Ned` itself calls the manager interface with a `config.json` describing a single container. The manager interprets the `config.json`, obtains the file-system data from the `rom/` or `rwfs/` namespace, and passes both to the runtime. The runtime implements the *create* and *start* operations of the OCI lifecycle, building the execution environment from the `config.json` and launching the application contained in the supplied file system. In this iteration a minimal viable set of `config.json` fields is parsed, the file-system layer is interpreted as a set of files in the static `rom/` namespace, and there are no resource-policy checks. The `config.json` itself is responsible for any required resource separation.

Iteration 2 — Storage, control, and full lifecycle: The Container Manager gains the ability to obtain file-system content from local storage, and its interface is extended to give `Ned` control to stop a container. The Container Runtime supports the full OCI lifecycle. The `config.json` may reference file-system layers held in local storage, each accompanied by a content hash.

Further iterations (potentially outside project scope): Subsequent iterations follow based on the learnings of the previous ones, progressively adding capabilities (for example richer interfaces, the CRI front-end, and VM-type container support) as priorities are confirmed.

7. Open Questions and Design Decisions

Several questions remain open and will be resolved during implementation. They are enumerated below for the benefit of reviewers and partners, allowing them to see the boundaries of the current design:

Host interface and versioning: Containers carry versions that depend on a specific host-interface version. This raises three connected questions:

- How can the system ensure that the host provides the interfaces a given container requires?
- What precisely constitutes the *host interface* of a container?
- Is explicit versioning of L4Re itself required to express compatibility?

In particular the last question is a matter of active L4Re development and its resolution is outside the scope of the project. While resolving these questions is a prerequisite for the portability and standards-conformance it is not required for the expected TRL of the prototype developed in the project.

Network layer: Whether containers expect a layer-2 or layer-3 network interface (Section 5.4) affects the device model the runtime must provide

Ned API: The programmatic interface for Ned (Section 5.5) is yet to be specified.

8. Summary and Conclusion

This deliverable has presented the initial design of container support for the L4Re Hypervisor and Operating System Framework. It surveyed the OCI container model and the standards the design adheres to and reviewed comparable systems, notably the manager/runtime split used by QNX and the VM-execution mechanism of Kata Containers. The deliverable then introduced a conceptual model that relates the portable OCI container to the explicitly provisioned L4Re compartment. From the project goals it proposed a working set of functional and quality requirements, and it specified an architecture in which a Container Manager and a Container Runtime, orchestrated initially by Ned, cooperate over IPC to instantiate containers inside appropriately sized L4Re compartments. The runtime maps each category of container resource onto an explicit L4Re mechanism, which is what allows the design to offer standards-aligned containers while retaining the platform's isolation guarantees, which are essential for a safety certification.

The central conclusion is that the OCI model is realisable on L4Re, and that the recommended path is incremental: a minimal Ned-driven prototype first, followed by the addition of local storage, stop control, and the full OCI lifecycle, with later iterations guided by the learnings each previous one produces.

9. Outlook

The immediate next step is the implementation of the minimal prototype described in **Iteration 1**, which will validate the manager/runtime separation and the resource-mapping approach in practice. Beyond that, work will proceed along two main items:

- i) Resolving the open questions of Section 7, foremost the host-interface and versioning model
- ii) and broadening the interface and execution options, including the Kubernetes CRI front-end and uvmm-based VM-type and Linux-compatibility execution

In line with the Objective O2 success criteria, the maturing runtime is to be demonstrated as a proof of concept in a relevant environment and evaluated by at least two independent testers, alongside tests confirming that containerised applications cannot exceed their allocated resources. These developments will be reported in the subsequent Task 2.2 deliverable D2.4 as the container middleware matures.