

POLITECNICO DI TORINO

Master's Degree in COMPUTER ENGINEERING



Master's Degree Thesis

Vehicular Architecture for the Orchestration of Containerized Vehicle Microservices

Supervisors

Prof. Fulvio RISSO

Ing. Domenico PERRONI

Ing. Massimiliano RASPANTE

Candidate

Luca FERRARIS

MARCH 2026

Ringraziamenti

Questa pagina l'ho voluta mettere all'inizio perché è la più importante tra tutte quelle presenti in questa tesi. È stato un percorso lungo e molte volte anche molto difficile, ma ho avuto la fortuna di avere attorno delle persone che mi hanno aiutato in un modo o nell'altro.

Grazie a Mamma e a Papà che durante tutti questi anni mi hanno accompagnato, insegnato moltissime cose e hanno sempre cercato di tenermi sulla strada giusta aiutandomi anche quando mi sentivo solo.

Grazie a Nonno Valentino che mi ha sempre scarrozzato da una parte all'altra (o mi ha prestato la macchina nei momenti in cui mi sembrava indispensabile) e mi ha fatto scoprire molti posti nuovi.

Grazie ad Andrea, mio fratello. Non siamo mai stati né uguali, come spesso ci descrivono, né mai siamo andati d'accordo (solo ogni tanto nei giorni dispari probabilmente). Nonostante questo in qualche modo mi sei stato sempre vicino, e non so se ti ricordi, mi hai fatto scoprire la bellezza dello sport che mi ha salvato in questi ultimi anni nei momenti più bui.

Grazie a Matteo, Alessandro ed Eleonora, che ad un certo punto hanno deciso di spuntare nella mia vita a caso e da allora continuiamo a fare a caso praticamente tutto, ma insieme. Mi avete fatto scoprire la bellezza delle montagne e tutto il mondo che ci ruota attorno e per questo non vi ringrazierò mai abbastanza.

Grazie a Davide, che ogni volta che gli propongo un modo nuovo per soffrire, lui è sempre lì ad accettare. È bello avere qualcuno con cui fare certe imprese un po' folli senza un motivo apparente.

Grazie a tutte le persone conosciute mentre arrampicavo, credo che non ci sia sport più stupido e allo stesso tempo stupendo e so che voi la pensate allo stesso modo.

E infine grazie ad Elisa, che in quest'ultimo anno mi è stata sempre accanto e mi ha spinto a fare cose di cui non pensavo di essere in grado. Mi ha sopportato nei momenti in cui ero ingestibile e mi ha fatto vedere che in realtà una soluzione c'era sempre.

Table of Contents

1	Introduction	1
1.1	Structure of the Thesis	1
1.2	Context	1
1.3	Problem	2
1.4	Goal	2
2	Background	3
2.1	Automotive software	3
2.1.1	Legacy E/E architecture	4
2.1.2	Limitations of domain-based architectures	5
2.1.3	Zonal architecture	6
2.1.4	Software advantages	7
2.1.5	Software Defined Vehicle	7
2.1.6	Vehicle Signal Specification	8
2.2	Containerization in automotive systems	8
2.2.1	Standard containerization technologies	9
2.2.2	Automotive-specific containerization challenges	9
2.3	Orchestration concepts	9
2.3.1	Cloud-native orchestration	10
2.3.2	Orchestration in distributed automotive and embedded systems	11
3	System Design	13
3.1	Design objectives	13
3.2	Architectural context	13
3.3	Orchestrator placement in the HPC	14
3.3.1	Centralized control model	14
3.3.2	Management of zonal controllers	14
3.4	Vehicle Data Abstraction	15
3.4.1	Purpose and scope	15
3.4.2	Relationship with the orchestration layer	16
3.5	Functional architecture overview	16
3.5.1	Workload layer	16
3.5.2	Container orchestration layer	17
3.5.3	Vehicle data access layer	17

3.6	Architectural rationale	17
3.7	Design implications	18
4	Orchestration Analysis	19
4.1	Orchestration landscape	19
4.1.1	Cloud-native orchestrators	20
4.1.2	Edge-oriented orchestrators	20
4.1.3	Architectural paradigms	21
4.2	Automotive and orchestration	22
4.2.1	Considerations for HPC and zonal systems	23
4.2.2	Functional requirements	23
4.2.3	Technical requirements	24
4.3	Selection methodology	24
4.3.1	Evaluation criteria	25
4.3.2	Candidates	25
4.4	Candidate architectures	26
4.4.1	k3s	26
4.4.2	BlueChi	29
4.4.3	Ankaio	32
4.5	Comparative evaluation	37
4.5.1	Evaluation methodology and test environment setup	37
4.5.2	Resource Consumption Analysis	39
4.5.3	Functional Capabilities	41
4.5.4	Discussion	42
4.6	Final selection and rationale	43
5	Implementation	44
5.1	System architecture	44
5.1.1	Deployment and bootstrap	45
5.1.2	Container runtime: Podman	46
5.1.3	Container networking	47
5.1.4	Network configuration management	47
5.1.5	KUKSA vehicle abstraction layer	48
5.2	Workloads	49
5.2.1	Definition and configuration model	49
5.2.2	Lifecycle operations	50
5.2.3	Dependency Management	54
5.3	Operational logging	54
5.4	Benchmarking and analysis tooling	55
5.4.1	Benchmark driver	55
5.4.2	Log analyser	56
5.5	Summary	56

6	Experimental Evaluation	58
6.1	Experimental setup	58
6.1.1	Virtual machine environment	58
6.1.2	Automotive hardware environment	59
6.2	Evaluation metrics	60
6.2.1	Operation latency	60
6.2.2	Startup time	60
6.2.3	Memory usage	61
6.2.4	Reliability and functional validation	63
6.2.5	Benchmarking methodology	64
6.2.6	Scope and limitations	64
6.3	Results and discussion	65
6.3.1	Results from the VM benchmark campaign	65
6.3.2	Results from the S32G board campaign	66
6.3.3	Cross-environment comparison	67
6.3.4	VSS data exchange, controlled relocation, and update recovery	67
6.3.5	Why the VM is still a useful benchmark platform	68
6.3.6	Summary	68
7	Conclusions	70
7.1	Achieved Objectives	70
7.2	Limitations	71
7.3	Possible Extensions	72
A	Supporting Code Listings and Analysis Environment Setup	73
	Bibliography	76

List of Figures

2.1	Zonal domain architecture overview, adapted from ETAuto [7]	5
2.2	Zonal vehicle architecture overview, adapted from ETAuto [7]	6
3.1	Centralized orchestration model within the HPC in a zonal vehicle architecture.	14
3.2	Layered architecture of the proposed SDV orchestration framework. The software stack runs on the HPC platform and interacts with vehicle signals through zonal controllers.	17
4.1	k3s architecture overview [11]	27
4.2	BlueChi architecture overview [12]	31
4.3	Ankaaios architecture overview [13]	33
4.4	Ankaaios Control Interface overview	33
4.5	Request path through the Ankaaios Control Interface	35
4.6	Memory usage per process for different orchestrators (logarithmic scale)	40
5.1	Deployment architecture of the Atlas framework within the Ankaaios cluster. The HPC node hosts the Ankaaios control plane and the Atlas management layer, while edge nodes run ank-agent instances responsible for managing containerized workloads through the Podman runtime.	45
6.1	VM benchmark: mean duration per workload and operation	61
6.2	VM benchmark: memory usage per workload	62
6.3	VM benchmark: overall outcome distribution	63

List of Tables

4.1	Orchestrator deployment strategies	38
4.2	Summary of evaluation methodology	39
4.3	Per-process CPU usage at idle with deployed workload.	39
4.4	Relative memory cost normalized to BlueChi as baseline.	41
4.5	Functional capabilities comparison across the three orchestrators. . .	41
6.1	Quantitative results from the retained S32G board log	66
6.2	Comparison between VM benchmark results and S32G board results	67

Acronyms

SDV	Software Defined Vehicle.
VSS	Vehicle Signal Specification.
CLI	Command-Line Interface.
OTA	Over-the-Air.
E/E	Electrical/Electronic.
ECU	Electronic Control Unit.
DCU	Domain Control Unit.
SDK	Software Development Kit.
OCI	Open Container Initiative.
OEM	Original Equipment Manufacturer.
OS	Operating System.
HPC	High-Performance Computing.
CAN	Controller Area Network.
ADAS	Advanced Driver Assistance Systems.
IPC	Inter-Process Communication.
TSN	Time-Sensitive Networking.

Chapter 1

Introduction

1.1 Structure of the Thesis

This thesis presents a structured approach to identifying and evaluating a lightweight orchestration solution for modern automotive software architectures. The objective is to improve software management and reduce the coupling between application software and underlying hardware by combining cloud-derived orchestration technologies with signal abstraction mechanisms commonly adopted in the automotive domain.

The work is structured around the following main steps:

- First, the requirements of the automotive domain are analysed to **identify** the characteristics that an **orchestration platform** should provide, particularly with respect to the limited hardware resources typically available in current vehicle infrastructures.
- Based on these requirements, three candidate orchestration platforms are evaluated: K3s [1], BlueChi [2], and Ankaaios [3]. The **comparison considers** factors such as system **footprint** and **functional capabilities**.
- Following the selection of the orchestrator, a **lightweight vehicle abstraction layer** based on open standards is implemented. In particular, the solution leverages the Vehicle Signal Specification (VSS) defined by COVESA [4] and OCI container images in order to enable the development and deployment of vehicle applications while remaining independent from specific automotive hardware and proprietary signal interfaces.

1.2 Context

The automotive industry is currently undergoing a major technological transition driven by several factors, including electrification, autonomous driving systems, over-the-air (OTA) software updates, and increased vehicle connectivity.

These developments have significantly increased the complexity of traditional distributed E/E (Electrical/Electronic) architectures used in legacy automotive

systems, making them increasingly difficult to maintain and update. At the same time, the requirements of automotive software are becoming closer to those of modern general-purpose computing systems such as personal computers, smartphones, and cloud platforms.

This evolution has led to the idea of managing automotive software using concepts inspired by modern software infrastructures, while still addressing the specific constraints of the automotive domain.

1.3 Problem

The number of electronic components and the amount of software deployed in modern vehicles are rapidly increasing in order to support advanced automotive functionalities. Even relatively simple vehicles may integrate dozens or even hundreds of Electronic Control Units (ECUs), each responsible for specific functionalities.

Traditionally, each ECU runs a largely monolithic software stack that may implement thousands of functions. This architectural approach makes system maintenance, software updates, and validation processes increasingly complex and difficult to manage.

1.4 Goal

The goal of this thesis is to evaluate and deploy an orchestration platform capable of managing containerized workloads in an automotive environment while maintaining acceptable performance and resource consumption.

Another important objective is to follow the architectural guidelines and standards commonly adopted in the automotive industry, ensuring that the developed framework remains compatible with realistic deployment scenarios.

Chapter 2

Background

2.1 Automotive software

The vehicle electronic architecture defines how electronic control units (ECUs), sensors, and actuators communicate and cooperate within a vehicle. In recent years, the discussion around E/E architecture topologies has gained significant attention and has undergone multiple evolutionary stages. [5]

The first stage, during the 1980s when the number of ECUs was still limited, was based on a **point-to-point** architecture. Each ECU was directly connected to other components through dedicated wiring. However, as the number of control units increased, this topology became unfeasible. The wiring complexity grew quadratically with the number of ECUs, following $\frac{N(N-1)}{2} = \Theta(N^2)$, and consequently, vehicle weight and production costs increased significantly.

The next step was the transition to a **bus-based architecture**, enabled by the introduction of the **CAN** protocol. In this paradigm, multiple ECUs share the same communication medium, significantly reducing the number of physical connections and improving system scalability. However, this approach introduced limitations in terms of available bandwidth and data throughput.

In the following years, the number and heterogeneity of vehicle functions, such as infotainment, advanced driver assistance systems (ADAS), and powertrain control, increased further. A single bus was no longer sufficient to meet these heterogeneous communication requirements. As a result, the **multi-bus** architecture was introduced. This model integrates multiple specialized network protocols alongside CAN and employs a **central gateway** to interconnect different buses and translate between protocols, enabling cross-network communication.

Despite these improvements, new challenges emerged, along with some persistent issues. These include excessive cable length, gateways acting as communication bottlenecks, and highly distributed and fragmented software systems. To mitigate these problems and reduce the overall number of ECUs, a functional paradigm

known as **domain-based architecture** was adopted, which currently represents the predominant automotive software architecture. This approach introduces Domain Control Units (DCUs), each responsible for managing a specific functional domain within the vehicle. While this model reduces the number of ECUs and increases computational power within domains, some structural limitations remain unresolved. These include the continued reliance on a central gateway, complex cross-domain communication, and still extensive and costly wiring harnesses.

To address the remaining limitations, a more radical paradigm shift has been proposed, moving from a function-oriented organization to a location-oriented one, known as **zonal architecture**.

2.1.1 Legacy E/E architecture

The current architecture adopted by most manufacturers is the so-called **domain-based architecture**. This architecture represents the evolution of the **multi-bus** structure, in which each specific function was associated with its own technological framework.

In this architecture, ECUs are no longer distributed across the vehicle for **single, specific functions**; instead, they are aggregated into main functional **domains**.

Typically, a vehicle is divided into five main domains [6]:

- Powertrain domain
- Advanced Driver Assistance Systems (ADAS) domain
- Infotainment domain
- Body electronic and lighting domain
- Passive safety domain

Each domain is managed by a **Domain Control Unit (DCU)** with significant computational capabilities, which coordinates the ECUs belonging to that domain.

ECUs exchange data over internal networks specific to their domain, while DCUs communicate across domains through a central **gateway**, which serves as a bridge between heterogeneous networks.

This architectural design has led to several benefits.

- Reduction in total number of ECUs
- Increased computational power within each domain
- Improved management of system complexity

From a communication perspective, networks remain heterogeneous, continuing to rely on multiple protocols such as CAN, Automotive Ethernet, and TSN-based technologies.

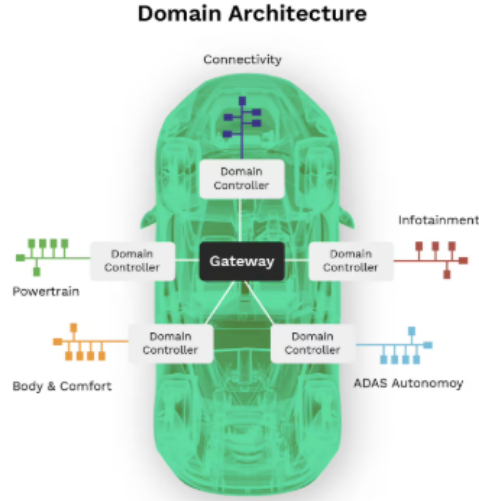


Figure 2.1: Zonal domain architecture overview, adapted from ETAuto [7]

Until now, this architecture has represented an effective compromise, enabling increased vehicle functionality without exponentially increasing system complexity and cost.

However, it retains an intrinsic separation between domains, which can create bottlenecks at the gateway level due to the growing number of cross-domain functions.

2.1.2 Limitations of domain-based architectures

This architecture presents structural limitations that become **critical** as software complexity and validation requirements increase.

Gateway and communications

The **gateway** concentrates all inter-domain communications within the vehicle. This inevitably increases latency and limits system scalability.

Modern in-vehicle functions require cooperation between multiple domains, leading to a significant increase in the volume of exchanged data. Since this data must pass through the gateway, it may result in communication bottlenecks and fragmented software integration.

Software fragmentation

The fragmentation described above stems from the fact that each domain typically adopts different technology stacks that are implemented and validated together with the underlying hardware. This tight coupling results in:

- Complex system integration
- Maintenance challenges
- Difficulty in keeping pace with modern software development practices

- Limited scalability of OTA updates

2.1.3 Zonal architecture

The **zonal architecture** organizes ECUs according to their physical location within the vehicle, shifting from a function-oriented segmentation paradigm to a geographically organized one.

This approach allows data from ECUs located in a specific area of the vehicle to be consolidated into a single **Zonal Controller**. These controllers forward aggregated data to a central **High-Performance Computing (HPC)** for processing, making the development of advanced software functions more scalable and practical.

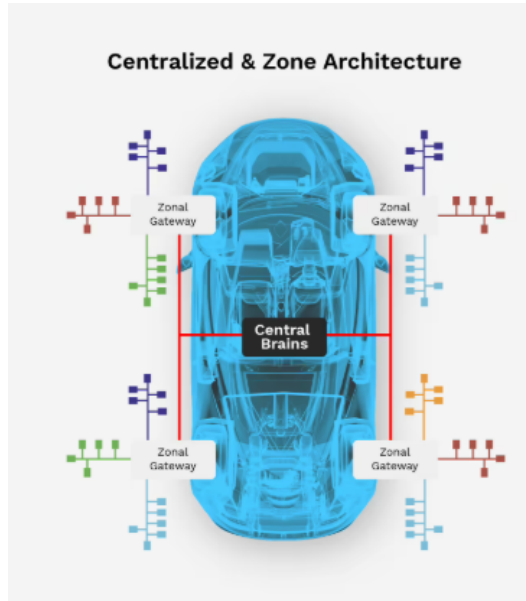


Figure 2.2: Zonal vehicle architecture overview, adapted from ETAuto [7]

Each zone typically includes sensors, actuators, and local ECUs. These elements are interconnected with the **zonal controller**, which transmits aggregated data to the central HPC via a high-speed backbone.

Common zonal divisions include:

- Front-left
- Front-right
- Rear-left
- Rear-right

This architectural shift introduces both hardware and software advantages.

Hardware advantages

The primary hardware improvement lies in the reduction of wiring length and the overall number of ECUs. This leads to:

- Lower vehicle weight
- Reduced production costs
- Simplified wiring harness design

Additionally, communication between different zones can occur through a unified high-speed backbone, typically based on Automotive Ethernet.

2.1.4 Software advantages

The software advantages stem from the ability to leverage this architectural design to abstract the hardware layer from the software layer.

This abstraction enables:

- Decoupling of the software lifecycle from the hardware lifecycle
- Improved support for OTA updates
- Centralized software management

These characteristics represent a fundamental step toward the **Software Defined Vehicle (SDV)** paradigm.

2.1.5 Software Defined Vehicle

Unlike traditional vehicles, where hardware dictates system capabilities, the **Software Defined Vehicle (SDV)** paradigm places software at the center of vehicle functionality. Hardware becomes a generalized execution platform upon which software-defined features are deployed, effectively decoupling vehicle functions from specific ECUs.

However, this paradigm can only be realized if the underlying E/E architecture supports centralization, high bandwidth communication, and hardware abstraction. The zonal architecture described above provides the structural foundation that enables this transition.

The key characteristics of the SDV include:

- Centralized computing platform
- Over-the-air (OTA) updates
- Function virtualization
- Continuous deployment
- Software standardization
- Cloud integration

In an SDV context, vehicle functions are increasingly implemented as software services running on shared computing resources. A common representation of vehicle signals becomes essential to enable cross-domain communication, service-oriented architectures, and cloud integration.

For this purpose, the **Vehicle Signal Specification (VSS)** provides a unified data model for describing vehicle signals in a structured and technology-agnostic way.

2.1.6 Vehicle signal specification

The **Vehicle Signal Specification (VSS)** is an open standard designed to define a **common data model and hierarchical signal tree structure** for representing vehicle data in a uniform, OEM-agnostic way.

Instead of dealing with vendor-specific data formats, VSS allows software developers to work with a consistent and hierarchical signal model.

```
Vehicle.Speed  
Vehicle.Cabin.Door.Row2.Right.IsOpen  
Vehicle.Powertrain.CombustionEngine.RPM
```

This structured approach further supports the separation between hardware and software layers by:

- Enabling standardization and interoperability across systems
- Facilitating cloud integration and service-oriented architectures

VSS is independent of the underlying communication protocol and can be mapped to different transport technologies.

2.2 Containerization in automotive systems

Containerization naturally fits within the context of the **zonal architecture** and the **Software Defined Vehicle (SDV)** paradigm. With the introduction of High-Performance Computing (HPCs) and a high-speed communication backbone, the vehicle no longer consists of a set of ECUs rigidly tied to specific functions, but rather of a **centralized computing platform** capable of running multiple software services.

In this context, containerization provides an execution model aligned with:

- Computing centralization
- Hardware–software abstraction
- Over-the-air (OTA) updates
- Separation of software and hardware life cycles

This enables multiple software services to run on a single platform. However, it also introduces new challenges related to isolation, service orchestration, and fault management.

2.2.1 Standard containerization technologies

Containerization is a form of **operating-system-level virtualization** that enables application **isolation** within lightweight environments called containers.

Unlike virtual machines, containers share the host operating system kernel, significantly **reducing overhead**. This characteristic makes them particularly suitable for automotive environments, where computational resources are constrained.

At a high level, the containerization model consists of:

- A **container image**, which packages the application and its dependencies
- A **container runtime**, responsible for executing and managing containers
- The **host operating system**, shared by all running containers

In the automotive context, the main benefits include service isolation, portability across environments, modular updates of individual functions, and the secure execution of heterogeneous applications on a centralized platform.

2.2.2 Automotive-specific containerization challenges

Despite its advantages, the adoption of containerization in automotive systems introduces significant challenges.

Unlike traditional cloud systems, automotive platforms must comply with stringent requirements, including:

- Deterministic behavior
- Functional safety standards
- Cybersecurity constraints
- Real-time execution requirements

Additional challenges include efficient resource allocation and strict control of CPU, memory, and I/O usage to guarantee predictable system behavior.

2.3 Orchestration concepts

Containerization enables application isolation and modular deployment, but it does not provide mechanisms for managing an increasing number of containers distributed across multiple zones or computing nodes.

As the number of software services running within the vehicle grows, an additional management layer becomes necessary to coordinate deployment and execution.

Orchestration technology addresses this need by providing automated mechanisms for:

- Deployment
- Scheduling
- Resource allocation
- Monitoring
- Scaling
- Fault recovery

These capabilities ensure that containerized applications are executed reliably, efficiently, and in accordance with predefined policies.

From a functional perspective, container orchestration can be described in three main phases:

1. **Provisioning:** defines the desired state of an application. This state is typically described in a declarative configuration file (YAML or JSON), specifying the container image, resource constraints, and networking requirements.
2. **Deployment and Scheduling:** the orchestrator determines the most suitable node within the system and deploys the container accordingly, taking into account resource availability and defined constraints.
3. **Lifecycle Management:** after deployment, the orchestrator continuously manages the container lifecycle. This includes health checks, restart policies, resource monitoring, and controlled updates.

[8]

In addition to these phases, modern orchestrators operate based on a **desired-state reconciliation model**, continuously comparing the current system state with the declared configuration and correcting deviations automatically.

2.3.1 Cloud-native orchestration

Container orchestration gained widespread adoption in cloud computing environments, where distributed systems composed of multiple computational nodes require automated lifecycle management at scale.

To address the complexity of large-scale distributed infrastructures, cloud-native orchestration platforms introduced several key capabilities:

- Declarative configuration models
- Microservices-oriented architectures

- Advanced resource management
- Horizontal scaling
- Service discovery and networking abstraction
- Rolling updates and rollback mechanisms

These features provide abstraction between software implementation and the underlying infrastructure on which applications run, enabling portability and elasticity across heterogeneous environments.

Cloud-native orchestrators are typically designed for clusters with abundant computational resources and flexible timing constraints, prioritizing scalability and availability.

2.3.2 Orchestration in distributed automotive and embedded systems

While cloud-native orchestration platforms are powerful, their direct adoption in embedded and automotive systems is not straightforward.

Automotive platforms differ significantly from cloud environments due to:

- Limited computational resources
- Deterministic timing requirements
- Functional safety constraints
- Certification requirements
- Strict power budget

Despite these constraints, several cloud-derived concepts are highly relevant in the context of Zonal Architectures and SDV platforms. Achieving effective hardware–software decoupling requires:

- Workload management
- Resource isolation and quota enforcement
- Health monitoring and fault recovery
- Controlled and traceable updates
- Service abstraction

However, these functions must be guaranteed together with additional embedded-system requirements, such as deterministic scheduling, mixed-criticality isolation, real-time constraint enforcement, controlled startup sequencing, and offline operation without cloud dependency.

Furthermore, automotive orchestrators typically operate on a limited number of nodes (e.g., one or a few HPCs), rather than large-scale elastic clusters. Therefore, scalability in this context refers more to workload consolidation and predictability than to horizontal expansion.

These constraints motivate the development of orchestration frameworks specifically tailored to the automotive domain, capable of combining cloud-inspired automation with embedded-system determinism and safety requirements.

Chapter 3

System Design

3.1 Design objectives

While container orchestration technologies are widely adopted in cloud environments, their direct integration into automotive platforms is not sufficient to satisfy the specific constraints of zonal architectures.

This chapter presents the conceptual system design for integrating an orchestration framework within an automotive environment. At this stage, no specific orchestration platform is selected. Instead, the focus lies on defining:

- The **architectural placement** of the orchestrator
- The **interaction** between the orchestrator, the High-Performance Computing (HPC), and the Zonal Controllers
- The introduction of a vehicle **data abstraction layer** based on VSS
- The **functional requirements** specific to automotive systems

The objective is to define a scalable and technology-agnostic orchestration architecture tailored to zonal vehicle platforms.

3.2 Architectural context

The proposed system builds upon a zonal E/E architecture in which computation is centralized within one or more High-Performance Computing (HPCs), while Zonal Controllers manage geographically distributed sensors and actuators.

In such a configuration:

- The HPC acts as the primary computational node
- Zonal Controllers provide localized I/O aggregation
- A high-speed backbone interconnects all nodes

Given this topology, the HPC represents the most suitable location to host the container orchestrator. Placing the orchestrator in the HPC allows centralized workload management while maintaining visibility and control over distributed zonal resources.

3.3 Orchestrator placement in the HPC

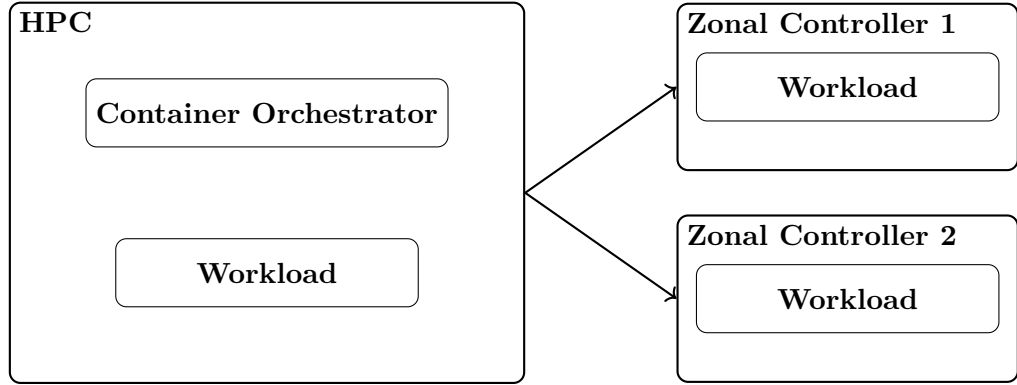


Figure 3.1: Centralized orchestration model within the HPC in a zonal vehicle architecture.

3.3.1 Centralized control model

The orchestrator is conceptually deployed within the HPC. From this position, the orchestrator can:

- Deploy and manage workloads on the HPC itself
- Communicate with Zonal Controllers
- Monitor system health across distributed nodes
- Coordinate software updates

This model ensures consistency with the centralized computing paradigm adopted by SDV platforms. Rather than having independent orchestration instances per zone, a unified control plane simplifies system validation and update strategies.

3.3.2 Management of zonal controllers

Zonal Controllers are treated as managed nodes capable of executing containerized workloads or application modules.

However, unlike cloud worker nodes, they operate under several constraints, including:

- Limited computational resources

- Strict real-time constraints
- Deterministic communication requirements

Therefore, the orchestrator must support controlled workload placement policies so that resource-constrained or timing-sensitive functions can be separated from less critical services.

3.4 Vehicle Data Abstraction

Container orchestrators provide the mechanisms required to deploy, schedule, and supervise software workloads across one or more computing nodes. However, they operate at the infrastructure level and do not define how applications should access vehicle data.

In automotive systems, software components often need to consume or publish information such as vehicle speed, battery level, steering angle, or diagnostic data. In practice, this information is generated by heterogeneous sources and exposed through bus-specific formats, ECU-specific interfaces, or proprietary software stacks. As a consequence, application logic can easily become tightly coupled to the underlying vehicle platform.

To address this issue, the proposed architecture introduces an **abstraction layer** positioned between vehicle applications and the underlying signal infrastructure. The purpose of this layer is to provide a stable and standardized representation of vehicle data, so that applications can interact with signals through a common model rather than through hardware-specific interfaces.

The abstraction layer is based on the **Vehicle Signal Specification (VSS)**, defined by COVESA, which represents vehicle data through a hierarchical and semantically meaningful signal tree. In the context of this work, VSS provides the common information model used to describe the signals exchanged by software components. A runtime technology such as **Eclipse KUKSA** [9] can then be used to expose and transport these signals in a uniform way, decoupling applications from the native vehicle communication details.

By relying on this approach, the system separates *how software is deployed* from *how vehicle data is represented and accessed*. This distinction is essential in a zonal architecture, where workloads may be distributed across different computing nodes while still requiring a coherent view of the same vehicle signals.

3.4.1 Purpose and scope

The abstraction layer simplifies the interaction between vehicle applications and the underlying signal infrastructure. In particular, it provides:

- Access to vehicle data through a **standardized signal model** based on VSS
- **Decoupling** between application logic and hardware-specific communication interfaces

- A **uniform access model** for software components that need to read or publish vehicle signals
- A **stable application-facing interface**, even when the native source of the data differs across platforms

This means that an application does not need to be written against CAN identifiers, ECU-local data structures, or proprietary middleware interfaces. Instead, it can be designed against logical signal paths defined in the VSS tree. The mapping from native vehicle data sources to these signal paths is handled below the application layer.

From a software-engineering perspective, this improves portability and modularity. Vehicle applications can be developed against a common signal vocabulary, while the adaptation to the specific vehicle platform is confined to the lower layers of the system.

3.4.2 Relationship with the orchestration layer

The abstraction layer and the orchestration layer address two distinct concerns and should not be conflated.

The **orchestration layer** is responsible for workload lifecycle management. It decides where workloads are deployed, starts and stops them, monitors their execution state, and maintains the desired system configuration.

The **abstraction layer**, instead, is responsible for the **representation and exchange of vehicle data**. It defines how applications refer to vehicle signals and how those signals are made available independently from the native hardware interfaces.

In other words, the orchestrator manages *where and when* software runs, whereas the abstraction layer defines *what vehicle data* that software sees and *how* it accesses it. Keeping these responsibilities separate makes the overall architecture easier to understand, evaluate, and evolve.

3.5 Functional architecture overview

The resulting system can be logically structured into three layers.

3.5.1 Workload layer

This layer contains the containerized applications implementing the vehicle functions. Typical workloads include infotainment services, data processing modules, diagnostic components, and other non-safety-critical software functions. By packaging these functionalities as containers, applications can be deployed, updated, and managed independently from the underlying hardware platform.

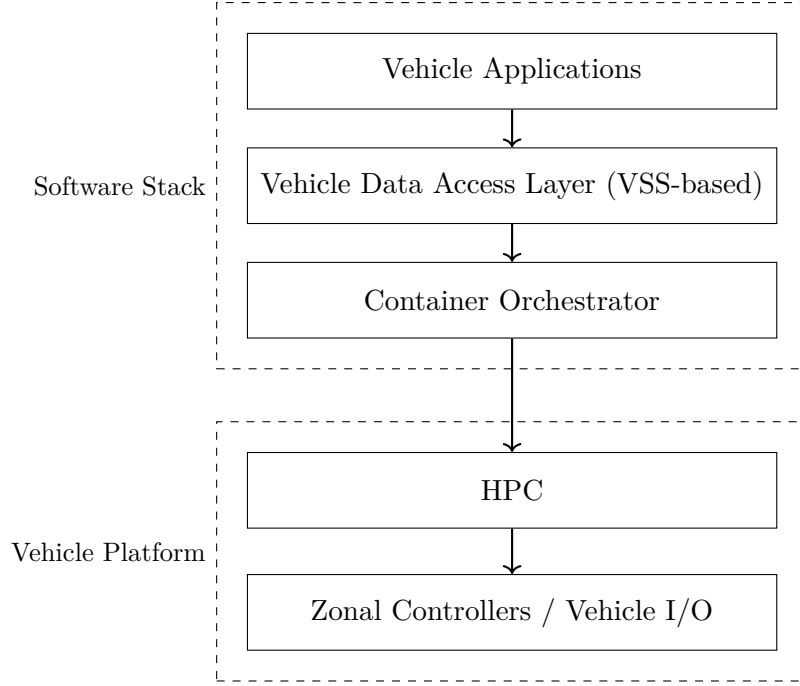


Figure 3.2: Layered architecture of the proposed SDV orchestration framework. The software stack runs on the HPC platform and interacts with vehicle signals through zonal controllers.

3.5.2 Container orchestration layer

The orchestration layer provides the mechanisms required to manage containerized workloads throughout their lifecycle. In particular, it is responsible for workload deployment, scheduling across available compute resources, restart handling, and execution-state supervision.

At a conceptual level, this layer acts as the execution management plane of the system. Its role is to ensure that the software deployed on the available nodes matches the intended system configuration.

3.5.3 Vehicle data access layer

The third layer of the proposed architecture is the vehicle data access layer. As introduced in Section 3.4, its purpose is to provide applications with a normalized and platform-independent view of vehicle signals.

Within the layered architecture, this component serves as the application-facing access point to vehicle signals and complements container orchestration by addressing a distinct concern: vehicle-data representation and access rather than workload execution.

3.6 Architectural rationale

A container orchestrator can manage the deployment and execution of distributed software components, but it does not by itself solve the problem of vehicle-data inter-

operability. In automotive systems, the same logical information may be produced by different ECUs, buses, or middleware components depending on the target platform.

For this reason, the proposed architecture complements workload orchestration with a dedicated vehicle-data abstraction. The orchestrator is responsible for where and when software runs, whereas the abstraction layer is responsible for how vehicle signals are represented and accessed by applications.

This separation of concerns improves modularity and software portability. This is the key reason why the proposed architecture treats workload orchestration and vehicle-data abstraction as complementary but distinct design concerns.

3.7 Design implications

The proposed architecture leads to several important implications for the deployment of containerized workloads in automotive environments:

- **Centralized orchestration** can be hosted on the HPC, which acts as the main coordination point for workload management
- **Distributed execution** remains possible across multiple nodes, including zonal controllers or other managed computing elements
- A clear **separation of concerns** is established between workload lifecycle management and vehicle-data access
- Applications can be developed against **standardized VSS signal paths** rather than hardware-specific interfaces
- The architecture supports a **modular software stack**, in which changes in the native data sources do not necessarily require changes in application logic

This conceptual design establishes the foundation for integrating container orchestration technologies into automotive environments while preserving a clean separation between infrastructure management and vehicle-data modelling.

Because the architecture remains focused on conceptual roles rather than on a specific implementation, the next chapter can evaluate candidate orchestrators independently from the VSS/KUKSA-based data abstraction introduced here. The orchestrator selection problem and the vehicle-data abstraction problem are related, but they are not the same problem and should be assessed separately.

Chapter 4

Orchestration Analysis

4.1 Orchestration landscape

The orchestration ecosystem has evolved substantially in the last decade and is characterized by heterogeneous solutions driven by the rapid adoption of cloud computing and microservice-based architectures. These platforms have become essential for deployment, scaling, and lifecycle management of containerized services.

However, the key concepts that led to the development of cloud orchestration platforms differ significantly from those emerging in edge architectures and automotive environments.

While cloud systems prioritize:

- Horizontal scalability
- Elasticity
- Multi-tenancy
- Fault tolerance
- Load balancing

Embedded domains, instead, emphasize:

- Determinism
- Resource efficiency
- Predictable behavior

For this research, it is therefore necessary to identify a solution that can primarily provide the functions required in the embedded and automotive domains, while also supporting cloud-oriented capabilities that meet the needs of the emerging SDV ecosystem.

4.1.1 Cloud-native orchestrators

Cloud-native orchestrators were designed to operate in large-scale data centers, where resources are virtually elastic, network connectivity is stable and high-bandwidth, and computational capacity is largely unconstrained.

Their objectives follow the principles listed above, including automated scaling, service discovery, **rolling updates**, and **declarative infrastructure management**.

These systems rely on a layered control-plane architecture composed of:

- API server
- Scheduler
- Control managers
- Distributed state storage

This subdivision provides the abstraction necessary to ensure that the management layer is decoupled from the application execution layer. The orchestration logic continuously **reconciles** the **desired system state** with the observed runtime state, implementing a declarative management paradigm.

All these features offer strong flexibility and ecosystem integration, but they introduce architectural complexity and non-negligible runtime overhead. Control-plane components and advanced networking abstractions (such as dynamic load balancing and service discovery) are fundamental in cloud environments but may be overdimensioned in embedded systems.

Examples of this category of orchestrators include Kubernetes (K8s), Docker Swarm, and Amazon ECS.

Moreover, recovery mechanisms in cloud-native orchestrators are often designed for availability rather than strict temporal determinism, which can represent a limitation in safety-critical automotive systems. [10]

4.1.2 Edge-oriented orchestrators

Edge computing has led to the development of lighter orchestration solutions tailored to resource-constrained and geographically distributed environments. Edge-oriented orchestrators aim to reduce the footprint of cloud-native stacks while preserving core lifecycle management capabilities.

Compared to cloud platforms, edge environments are characterized by:

- Limited computational and memory resources
- Heterogeneous hardware architectures
- Intermittent or bandwidth-constrained connectivity
- Reduced operational staff and simplified management requirements

Edge orchestrators therefore simplify the control plane, reduce external dependencies, and often integrate more tightly with the underlying operating system. In some cases, they leverage native Linux service managers to coordinate distributed workloads across multiple nodes.

Examples of orchestrators designed for these environments include K3s, KubeEdge, and Azure IoT Edge.

These solutions represent an important step toward lightweight orchestration; however, they are typically designed for industrial IoT or distributed computing scenarios. As such, they may not explicitly address automotive-specific constraints, such as strict startup-time requirements, functional-safety integration, and vehicle-level architectural hierarchies.

Beyond the categorization based on deployment domain, orchestration systems can also be analyzed through their architectural paradigms. These paradigms define how control responsibilities are distributed, how the state is managed, and how interaction with the system is modeled.

4.1.3 Architectural paradigms

Container orchestration systems are structured around a fundamental architectural separation between the **control plane** and the **data plane**. The control plane is responsible for maintaining the desired state of the system, making scheduling decisions, and coordinating workload lifecycle operations. The data plane, instead, executes the workloads and provides the runtime environment in which applications operate.

Centralized vs Distributed Control Architectures

In a **centralized control-plane architecture**, a single logical entity maintains global visibility over the cluster state and performs orchestration decisions. Worker nodes execute workloads but do not autonomously determine scheduling or placement policies. This model simplifies governance and ensures coherent and simplified resource management. However, it may represent a single point of failure if not properly replicated.

On the other hand, **distributed orchestration architectures** decentralize decision-making. Multiple nodes may participate in state management and scheduling, often relying on distributed consensus algorithms to maintain consistency. This approach enhances scalability and fault tolerance, but increases complexity and communication overhead.

While distributed models are well-suited for large-scale cloud infrastructures, their benefits must be carefully evaluated in environments where the number of nodes is fixed, the topology is known in advance, and resource constraints are more stringent.

Hierarchical orchestration models

Between purely centralized and fully distributed paradigms lies the **hierarchical orchestration model**. In this approach, orchestration responsibilities are organized across multiple layers. A higher-level controller supervises subordinate domains, each potentially managing its own local execution environment.

Declarative vs imperative orchestration

Another fundamental distinction in orchestration paradigms concerns the interaction model between users and the system: **imperative** and **declarative** management.

In imperative orchestration, users provide explicit procedural instructions describing how the system should transition from one state to another. Operations are executed sequentially, and the outcome depends on the order and correctness of the commands issued. While this approach offers fine-grained control, it complicates recovery mechanisms and reduces automation capabilities.

Declarative orchestration, instead, relies on the specification of a desired system state. The orchestrator continuously observes the actual runtime conditions and performs reconciliation actions to align them with the declared configuration. This model enables self-healing behavior, idempotent operations, and improved reproducibility. It is widely adopted in modern cloud-native platforms due to its robustness and automation potential.

However, the reconciliation mechanisms intrinsic to declarative systems may introduce non-deterministic timing behavior, especially when automatic retries, dynamic scaling, or reactive restarts are involved.

4.2 Automotive and orchestration

The transition toward Software-Defined Vehicles and High-Performance Computing platforms has introduced new challenges in software deployment and lifecycle management within vehicles. Modern automotive architectures increasingly adopt zonal designs, where a central HPC coordinates multiple zonal controllers responsible for specific physical areas of the vehicle.

In this context, orchestration systems must operate under constraints that differ from both cloud and generic edge environments:

- Strict memory and CPU limitations compared to data centers
- Deterministic and predictable startup behavior
- Offline operability without reliance on external cloud services
- Tight integration with in-vehicle communication stacks
- Security-by-design principles aligned with automotive standards

These characteristics overlap partially with the objectives of edge-oriented orchestrators. However, automotive infrastructures also require capabilities traditionally associated with cloud platforms, such as centralized lifecycle governance and coordinated software updates across multiple computational domains.

4.2.1 Considerations for HPC and zonal systems

When considering an HPC-based vehicle architecture with distributed zonal controllers, the placement of the orchestrator becomes a key design decision. A centralized control-plane model, hosted on the HPC, simplifies system governance and enables consistent lifecycle management across the vehicle. Zonal nodes operate as managed execution environments under the supervision of the central orchestrator. Such a hierarchical yet centrally governed model differs significantly from large-scale cloud clusters, as it operates within a closed and predefined network topology, without dynamic node admission, horizontal auto-scaling, or multi-tenant resource competition.

Therefore, the selection of an orchestrator for automotive HPC platforms requires a careful evaluation of architectural overhead, runtime footprint, deterministic behavior, and alignment with vehicle-specific constraints.

4.2.2 Functional requirements

SDV transforms the vehicle into a continuously evolving software platform. Functionality is no longer statically bound to Electronic Control Units (ECUs), but can be deployed, updated, and reconfigured over time. Consequently, the orchestrator must provide a set of functional capabilities aligned with this new software-centric model.

- **Deterministic workload deployment:** automotive systems require deterministic and reproducible workload instantiation. Additionally, non-safety-critical applications must be started in a predefined sequence. The orchestrator needs to support explicit dependency management and controlled startup ordering across the different nodes (HPCs and zonal controllers).
- **OTA update support:** modern vehicles require controlled over-the-air software updates to **reduce maintenance effort** and avoid dealership visits for software changes. In the automotive domain, update mechanisms must **preserve system stability**, support validation before activation, and **provide rollback** capabilities in case of failure.
- **Resource-aware scheduling:** vehicles operate within fixed hardware boundaries. Orchestrators must be capable of assigning workloads while respecting hardware constraints in terms of CPU, memory availability, and real-time priorities. These features become essential to prevent the performance degradation of timing-sensitive or higher-criticality functions.
- **Inter-node communication:** the **zonal architecture** introduces distributed workloads spanning HPCs and zonal controllers. Orchestrators must ensure

efficient inter-node communication.

4.2.3 Technical requirements

Automotive High-Performance Computing (HPC) platforms represent a significant technological evolution, but their hardware capabilities remain constrained when compared with cloud servers. Platforms such as the NXP S32G family integrate multi-core ARM Cortex processors, networking and security accelerators, and automotive-grade peripherals. However, they still operate under strict power, thermal, and memory constraints, and typically provide only a limited number of CPU cores and a few gigabytes of RAM. As a result, the technical requirements for orchestration in this context differ significantly from those of cloud infrastructures.

- **Low memory footprint:** HPCs provide limited RAM resources compared to enterprise servers, typically 2–4 GB of memory. Therefore, it is of primary importance to reduce memory consumption as much as possible. Avoiding unnecessary components and background services is therefore essential
- **Fast startup time:** Vehicle systems must reach operational readiness within tightly bounded time intervals. These platforms require predictable and rapid boot sequences.
- **Fault tolerance** with predictable behavior: Orchestrators must provide fault-handling mechanisms that remain predictable and bounded in time
- **Safety and security compliance:** The orchestration layer must support traceability, configuration integrity, and controlled lifecycle management compatible with automotive development processes
- **Offline operability:** Vehicles must remain fully functional without permanent connectivity to external cloud services. The orchestrator must operate autonomously within the in-vehicle network, without reliance on external registries, control servers, or remote state stores.

Based on the functional and technical requirements identified above, a set of orchestration platforms was selected for further evaluation. The following section introduces the candidate platforms considered in this research and motivates their inclusion in the comparative analysis.

4.3 Selection methodology

Based on the orchestration landscape analysis and on the functional and technical requirements identified in the previous sections, a selection methodology is required to determine which orchestration framework is most suitable for the automotive context considered in this work.

The objective of this phase is not to determine the “most feature-rich” solution, but rather the platform that best aligns with the architectural constraints and operational expectations of Software-Defined Vehicles. The evaluation, therefore, focuses on compatibility with deterministic behavior, constrained hardware environments, and hierarchical HPC–zonal system topologies.

The selection process follows three main steps:

1. Definition of evaluation criteria derived from automotive functional and technical requirements;
2. Identification of representative orchestrators across different architectural categories;
3. Comparative assessment based on architectural characteristics, runtime behavior, and ecosystem maturity.

4.3.1 Evaluation criteria

The evaluation criteria are derived directly from the constraints and expectations discussed in the previous sections:

- **Architectural suitability for HPC + Zonal Controllers:** ability to support a centralized or hierarchical control-plane model compatible with an HPC supervising distributed zonal nodes.
- **Runtime footprint:** memory and CPU overhead introduced by control-plane components and supporting services, with particular attention to resource-constrained automotive hardware.
- **Startup performance:** initialization latency of the control plane and managed workloads, considering the strict boot-time requirements of in-vehicle systems.
- **Deterministic behavior:** degree of control over scheduling, restart policies, and reconciliation mechanisms.
- **Maturity and ecosystem support:** community adoption, documentation quality, long-term maintainability, and industrial backing.

These criteria enable a balanced comparison of cloud-derived and edge-oriented orchestrators while accounting for the specific requirements of automotive systems.

4.3.2 Candidates

Based on the previously defined criteria, the analysis excludes orchestrators that are excessively complex for the target automotive scenario or that do not provide the minimum features required for distributed workload management. Three candidate platforms have been selected for the comparative analysis:

- **K3s**
- **BlueChi**
- **Ankaaios**

These solutions represent three distinct orchestration philosophies:

K3s is a lightweight Kubernetes distribution designed for edge and resource-constrained environments. It preserves the declarative and API-driven model of Kubernetes while reducing the footprint by removing non-essential components and consolidating control-plane services. As a cloud-derived but edge-optimized solution, it represents a pragmatic adaptation of cloud-native orchestration to constrained environments.

BlueChi is a systemd-based orchestration framework targeting edge and embedded Linux environments. Instead of introducing a full container orchestration stack, it coordinates services across multiple nodes using native Linux service management primitives. This approach emphasizes simplicity, tight OS integration, and reduced overhead, making it an interesting candidate for deterministic automotive scenarios.

Ankaaios is a workload orchestrator designed specifically for embedded and automotive platforms. It follows a centralized server-agent model and supports distributed execution across multiple nodes. In the implementation used in this thesis, Ankaaios was evaluated with the **podman** runtime and a YAML startup manifest, which is also the execution model reused later in the Atlas prototype.

The following sections provide a detailed comparative analysis of these candidates against the defined evaluation criteria.

4.4 Candidate architectures

4.4.1 k3s

Architecture overview

K3s is a fully CNCF-compliant Kubernetes distribution specifically designed for lightweight and resource-constrained environments. While preserving the core Kubernetes control-plane functionalities, it introduces several optimizations that make it suitable for edge and embedded systems.

One of the main design choices is its distribution as a single binary, which significantly simplifies installation, deployment, and maintenance in environments with limited computational and storage resources. Despite its reduced footprint, k3s retains the fundamental Kubernetes abstractions and APIs, ensuring compatibility with standard Kubernetes manifests and tooling.

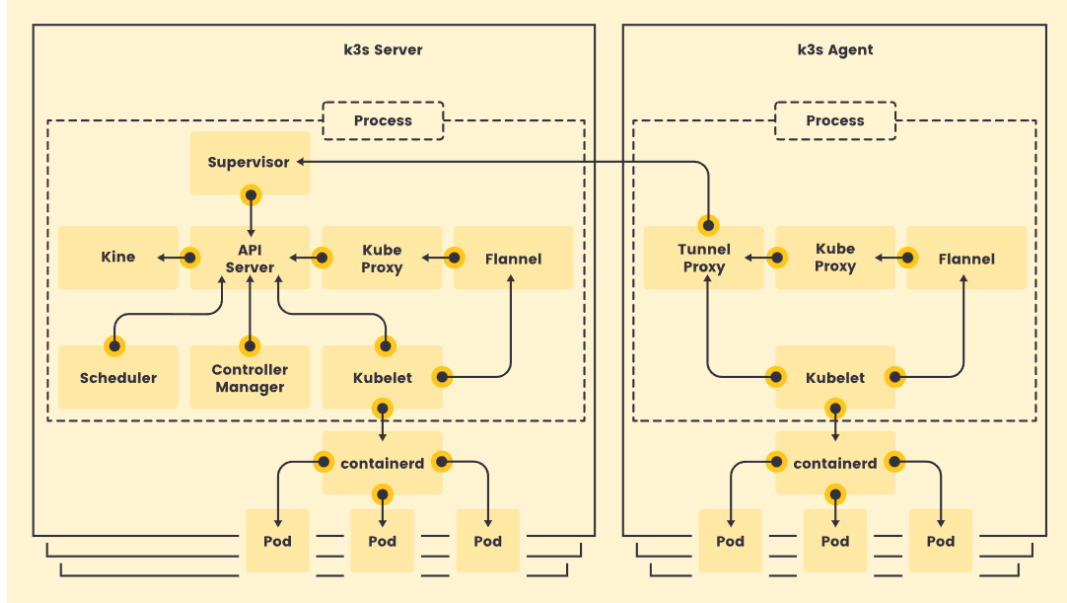


Figure 4.1: k3s architecture overview [11]

Architecturally, k3s follows a **Server-Agent** pattern. The server component hosts the control plane, while one or more agents act as worker nodes. The architecture can be deployed either in a multi-node configuration, where the server and agents run on different machines, or in a single-node setup where both roles coexist on the same device.

Server A **server node** runs the **k3s server** process and includes the control-plane components as well as the datastore. Its main components are:

- **API Server:** handles REST requests and manages the overall cluster state.
- **Scheduler:** assigns workloads (Pods) to registered nodes based on resource availability and scheduling policies.
- **Controller Manager:** runs the controllers responsible for maintaining the desired cluster state.
- **Datastore:** by default, k3s uses an embedded SQLite datastore through Kine in single-node deployments; in high-availability configurations, embedded etcd can be used.
- **Kubelet:** manages local pod lifecycle on the server node.
- **Container runtime:** typically containerd, responsible for executing containers.
- **CNI plugin:** Flannel is used by default to provide pod networking.
- **Load balancer:** a lightweight built-in load balancer (Klipper LB) is included for service exposure.

Agent An **agent node** is responsible for physically executing the workloads scheduled by the server. It runs a reduced set of components:

- **Kubelet:** manages the lifecycle of pods on the node.
- **Container Runtime:** the same runtime used on the server (typically containerd).
- **CNI Plugin:** provides networking capabilities for deployed pods.
- **Tunnel Proxy:** enables secure and persistent communication between agents and the server, ensuring stable connectivity with the control plane.

Storage and workload definition From a storage perspective, k3s follows the standard Kubernetes model based on **Persistent Volumes (PV)** and **Persistent Volume Claims (PVC)**. Storage provisioning and integration with external storage systems are handled through the **Container Storage Interface (CSI)** and, when required, the **Cloud Provider Interface (CPI)**. This abstraction layer enables workloads to remain portable across different infrastructures without requiring configuration changes.

Workloads are defined using declarative YAML manifests, identical to standard Kubernetes resources. These manifests can be applied using the same tooling (e.g., `kubectl`), ensuring interoperability between cloud and edge deployments. This compatibility is particularly relevant in hybrid scenarios where similar workloads must be deployed across both centralized cloud clusters and edge devices.

Strengths

K3s offers several strengths that make it well-suited for edge and resource-constrained environments.

First, it is fully compliant with the Kubernetes API, ensuring compatibility with the vast ecosystem of cloud-native tools and workloads. This compatibility enables seamless portability between cloud and edge deployments, allowing the same manifests, CI/CD pipelines, and operational practices to be reused across heterogeneous infrastructures.

Second, its lightweight design significantly reduces the operational footprint compared to standard Kubernetes distributions. The single-binary packaging simplifies installation and maintenance, while the removal of non-essential cloud-specific components minimizes resource consumption. This makes K3S particularly attractive for embedded systems and edge devices with limited CPU and memory availability.

Another relevant strength is its integrated approach: built-in networking (Flannel), embedded datastore options, and a lightweight load balancer reduce the need for external dependencies. This simplifies cluster setup and lowers operational complexity, especially in small-scale deployments.

Finally, K3s benefits from the maturity, stability, and large community support of the Kubernetes ecosystem. This translates into robust documentation, frequent

updates, and long-term maintainability — aspects that are particularly relevant for industrial and production environments.

Limitations

Despite its advantages, K3s inherits several structural characteristics from Kubernetes that may represent limitations in specific domains such as automotive systems.

One primary limitation is the complexity of the Kubernetes control-plane model. Even in its reduced form, k3s relies on multiple control components, reconciliation loops, and asynchronous state convergence mechanisms. While effective in cloud environments, this model may introduce non-deterministic behaviors that are not always aligned with the strict timing and predictability requirements of automotive workloads.

Additionally, Kubernetes-based orchestration is inherently designed for highly dynamic, multi-tenant cloud environments. Features such as service abstraction, dynamic scheduling, and network virtualization introduce abstraction layers that may add overhead in scenarios where system topology is relatively static and predefined, as often happens in in-vehicle high-performance computing (HPC) systems.

Resource consumption, although reduced compared to full Kubernetes, can still be significant for deeply embedded platforms. The presence of multiple background components and continuous state reconciliation processes may impact memory footprint and startup time.

Finally, k3s does not natively provide automotive-specific features such as deterministic startup sequencing, fine-grained workload dependency control tailored to embedded constraints, or tight integration with vehicle communication buses. Achieving such behavior typically requires additional configuration or external mechanisms, increasing system complexity.

These limitations do not diminish the robustness of k3s as a general-purpose edge orchestrator; however, they highlight potential challenges when targeting highly constrained automotive environments with strict timing and resource constraints.

4.4.2 BlueChi

Architecture overview

BlueChi is a **deterministic multi-node service controller** designed for embedded and edge environments. Unlike cloud-derived orchestrators, BlueChi builds upon native Linux technologies, particularly **systemd**, to extend service management across multiple nodes.

BlueChi integrates with systemd via the **D-Bus API** on each node and enables multi-node coordination by relaying control messages over **TCP**. Through this mechanism, it extends systemd’s local service management capabilities into a distributed environment while preserving deterministic service behavior.

The project is developed within the Eclipse Software Defined Vehicle (SDV) Working Group [**eclipseSDVwg**], reflecting a strong alignment with automotive and

embedded system requirements. This aspect is particularly relevant in the context of automotive platforms, where deterministic behavior and tight integration with Linux-based systems are essential.

Among the orchestrators analyzed, Eclipse BlueChi operates at the level of the underlying init and service manager (systemd). Rather than managing containers directly, BlueChi controls and monitors systemd unit states across nodes using a controller–agent architecture. Consequently, workloads must typically be encapsulated within systemd units (e.g., services) that define how processes or containers are executed. Compared to container-centric orchestrators, this introduces an additional integration step during deployment.

BlueChi follows a controller–agent architecture, complemented by a command-line interface:

- **Controller** (*bluechi-controller*): runs on the central node and manages the configuration and coordination of all registered agents.
- **Agent** (*bluechi-agent*): runs on each managed node and communicates locally with systemd over D-Bus to control and monitor service units.
- **Command-Line interface** (*bluechictl*): provides administrative access for testing, debugging, and manual interaction with the system.

Agents register with the controller and expose the state of local systemd units. The controller can then coordinate service operations across nodes, respecting dependencies and propagating start, stop, or restart commands in a controlled manner.

A fundamental architectural characteristic of BlueChi is that it does not act as a global state manager. Instead, it assumes that an external component is responsible for defining the desired system state. BlueChi provides the mechanisms to transition services between states (e.g., start, stop, restart) and to report their status, but it does not autonomously determine or enforce the overall target configuration.

This design has several implications. First, BlueChi focuses on service lifecycle execution rather than declarative state reconciliation. Second, while it reports service state changes and failures, it does not implement high-level recovery policies. Finally, it assumes that the system boots into a predefined configuration and does not manage full cluster initialization logic.

Thanks to its use of the local D-Bus system bus, BlueChi can be integrated with other Linux-based applications without introducing additional abstraction layers or runtime overhead.

Strengths

BlueChi presents several strengths that make it particularly suitable for embedded and automotive environments.

First, it is built directly on top of **systemd**, which is the de facto standard init and service manager in modern Linux-based systems. This tight integration

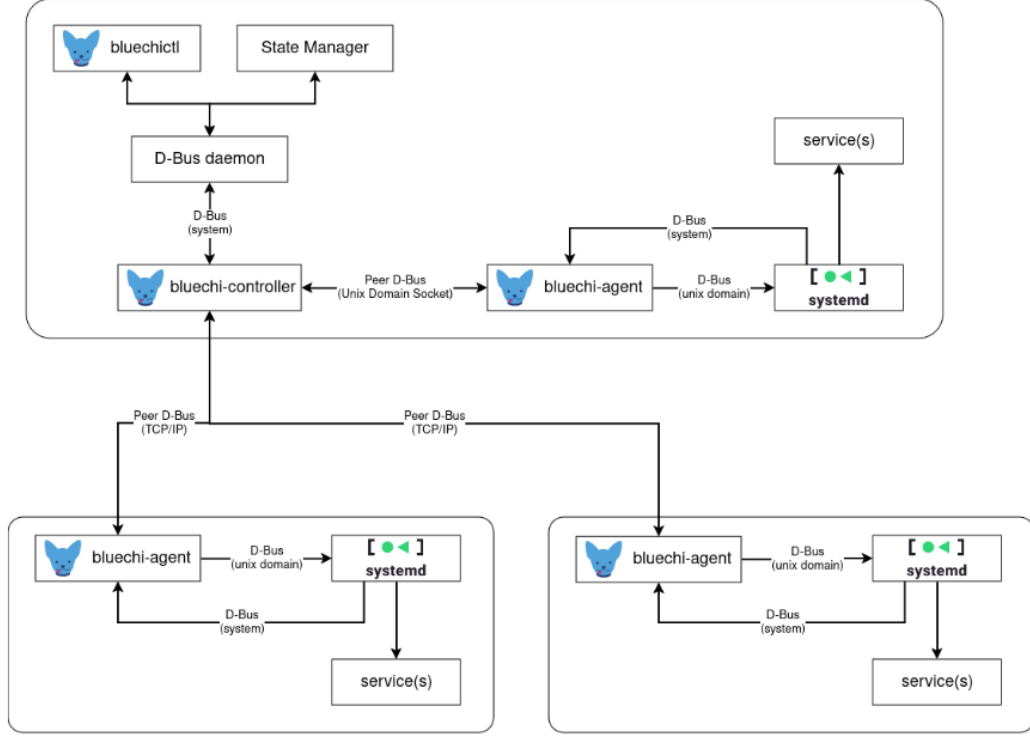


Figure 4.2: BlueChi architecture overview [12]

allows BlueChi to leverage native service supervision, dependency management, and deterministic startup mechanisms already available in the operating system.

Second, its architecture is lightweight and does not introduce heavy abstraction layers. By extending systemd through D-Bus communication and a controller–agent model, BlueChi enables multi-node coordination without replicating the complexity of cloud-native control planes. This makes it particularly attractive for resource-constrained edge devices.

Another important strength is its deterministic behavior. Since BlueChi operates at the service level and relies on systemd’s execution semantics, service transitions (start, stop, restart) follow well-defined and predictable rules. This is especially relevant in automotive systems, where timing predictability and controlled state transitions are critical.

Furthermore, BlueChi aligns closely with Linux-based embedded workflows. It does not impose a new declarative model or container abstraction but instead integrates with existing service definitions. This reduces conceptual overhead for teams already familiar with systemd-based system management.

Finally, being developed within the Eclipse Software Defined Vehicle ecosystem, BlueChi is designed with automotive use cases in mind, favoring simplicity, determinism, and integration over cloud-scale dynamic orchestration.

Limitations

Despite its advantages, BlueChi exhibits several structural limitations when compared to higher-level orchestration frameworks.

A primary limitation is that BlueChi is not built around declarative desired-state reconciliation. Instead, it provides distributed service-control primitives for managing systemd units across nodes. This keeps the architecture simple and deterministic, but it also means that high-level system-state management must be handled by surrounding tooling or by the service definitions themselves.

Closely related to this aspect, BlueChi does not provide a cloud-style scheduling layer or an automatic placement mechanism. It assumes that the relevant services and their unit files are already defined on the managed nodes and that lifecycle operations are triggered explicitly through its controller and D-Bus APIs.

Another limitation concerns container-based deployments. BlueChi does not treat containers as first-class orchestration objects. When Podman-based containers are used, they are typically managed indirectly through systemd units or quadlets. This approach is fully viable, but it shifts container lifecycle modeling to systemd rather than to a dedicated container orchestrator.

Moreover, BlueChi does not provide advanced orchestration features such as dynamic scheduling, automatic workload placement, rolling updates, or self-healing policies. Its focus remains on deterministic service control rather than cloud-style elasticity and automated reconciliation.

These limitations do not represent design flaws; rather, they reflect a deliberate architectural choice. BlueChi prioritizes deterministic service management and tight integration with Linux systems over high-level declarative orchestration capabilities.

4.4.3 Ankaaios

Architecture overview

Ankaaios is a lightweight workload orchestrator designed for distributed automotive and edge environments. It follows a server-agent architecture that provides a central point for managing software workloads across one or more nodes. In a zonal automotive platform, the server can run on the HPC, while agents are deployed on the nodes that execute the workloads.

Each Ankaaios deployment is composed of two main executables:

- **ank-server**: the central control-plane component. At startup, the server loads the cluster startup manifest and stores it as the **desired state**. It then instructs the agents to start or stop workloads in order to converge the system towards that state. Each cluster runs exactly one server instance, which acts as the single source of truth for the cluster state.
- **ank-agent**: the node-level component responsible for executing workloads on a given node. An agent runs on every node where workloads must be started

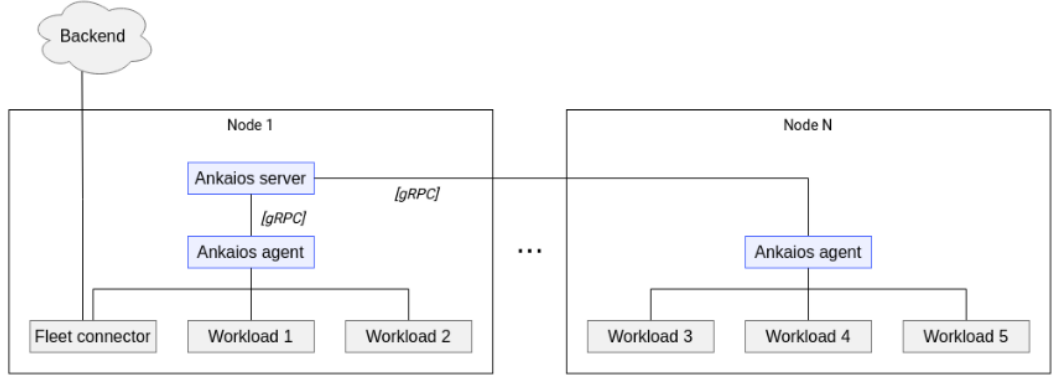


Figure 4.3: Ankaaios architecture overview [13]

and stopped. The agent receives commands from the server and applies them through the configured runtime. Since the server does not execute workloads directly, an agent must also run on the same node whenever workloads are to be hosted there.

From an operational perspective, Ankaaios separates cluster coordination from workload execution. This keeps the control plane compact and makes the architecture scalable from single-node setups to distributed deployments with multiple execution nodes. Communication between the server and the agents currently relies on **gRPC**. The project documentation also notes that this middleware is interchangeable in principle, although gRPC is the protocol used by the current implementation.

Another relevant design decision is that **inter-workload communication is outside the scope of Ankaaios**. The orchestrator manages workload lifecycle and cluster state, but application-level communication must be implemented separately through technologies such as network communication, IPC, shared memory, or other middleware chosen by the system integrator.

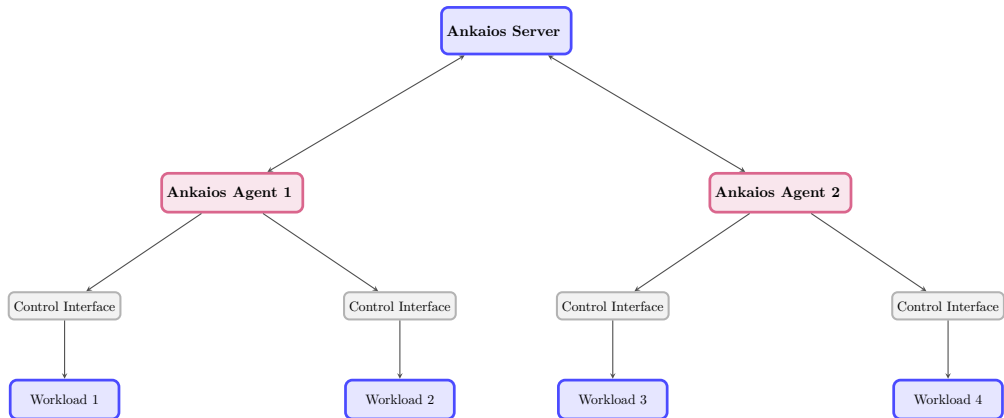


Figure 4.4: Ankaaios Control Interface overview

Workload model

In Ankaaios, a workload is any software unit whose lifecycle can be managed by the orchestrator, for example by starting, stopping, or restarting it. According to the official terminology, a workload can be packaged either as an **OCI container** or executed as a **native workload**. This makes the system extensible across different runtime back ends and allows it to support both containerized applications and software that must still run directly on the host platform.

The desired state of the system is described through a YAML-based manifest. For each workload, the manifest specifies the execution node, the selected runtime, the restart policy, optional tags and dependencies, and the runtime-specific configuration required to start the application. The server stores this information as the desired cluster state, while the agents are responsible for enforcing it on their respective nodes.

A key aspect of this model is that Ankaaios focuses on **workload lifecycle management**, not on service composition or service-to-service communication. Each workload is managed independently according to the desired state, whereas data exchange between workloads must be implemented through external mechanisms provided by the underlying platform.

Control interface

The Control Interface is the mechanism through which a workload managed by Ankaaios can communicate with the orchestration system. This interface enables a workload to request information about the cluster state or to trigger controlled state changes, thereby supporting operator-style patterns implemented as workloads within the cluster.

Communication through the Control Interface is mediated by the **ank-agent** managing the workload. Requests issued by the workload are first received by the local agent, checked against the configured authorization rules, and then forwarded to the server if they are allowed. This design keeps the server protected from arbitrary workload actions while still enabling controlled runtime interaction.

At runtime, the interface is exposed inside the workload through a mount point located at `/run/ankaaios/control_interface/`. The workload interacts with two FIFO files, **input** and **output**, which are used to exchange messages with the Ankaaios system. Messages are serialized using **Protocol Buffers**, and Ankaaios provides the corresponding protocol definitions so that client code can be generated in different programming languages.

The exchanged messages use a **length-delimited protobuf encoding**, which allows multiple serialized messages to be transmitted through the same FIFO stream while preserving message boundaries. This results in a lightweight, asynchronous communication mechanism that does not require direct network calls from the workload to the control plane.

It is worth noting that the Control Interface is currently available for workloads

using the `podman` runtime, while it is not provided for the `podman-kube` runtime.

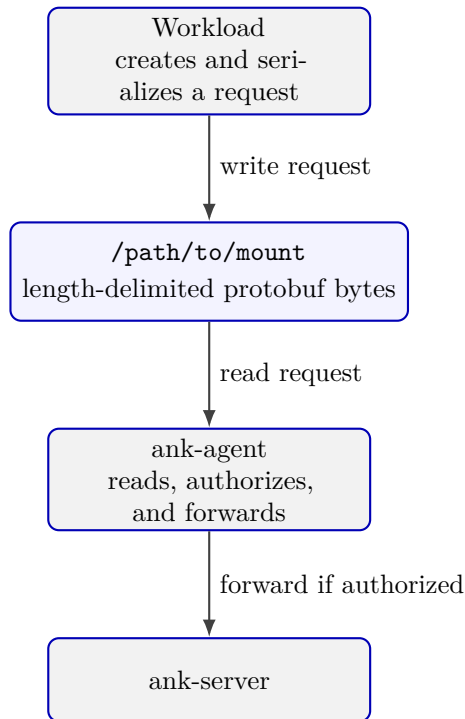


Figure 4.5: Request path through the Ankaaios Control Interface

Strengths

Ankaaios presents several characteristics that make it a strong candidate for automotive and edge-oriented deployments.

Lightweight architecture Ankaaios adopts a compact server-agent architecture with a limited number of core components. Compared with cloud-native orchestrators, this reduces operational complexity and resource consumption while still providing the essential mechanisms needed to manage workload lifecycles across multiple nodes.

Clear separation between control and execution The server is responsible for cluster coordination and desired-state management, while the agents are responsible for workload execution on the target nodes. This separation keeps the control plane simple and makes the deployment model easy to understand.

Support for multiple workload types Ankaaios can manage both OCI containers and native workloads. This is particularly useful in automotive environments, where new containerized services may need to coexist with existing host-level applications.

Declarative desired-state model The orchestrator uses a declarative YAML manifest to describe the intended state of the system. This simplifies deployment, reproducibility, and controlled reconfiguration of the software stack.

Workload-level integration through the Control Interface The Control Interface allows managed workloads to interact with the cluster in a structured way. Because it relies on FIFO files and protobuf-based messages, it remains lightweight and language-agnostic while supporting controlled runtime reconfiguration patterns.

Good fit for distributed automotive platforms Ankaaios is designed around distributed execution on multiple nodes and is therefore well aligned with centralized-compute and zonal architectures, where software components may run on different controllers under a common orchestration strategy.

Limitations

Despite its advantages, Ankaaios also has limitations that must be considered in the comparison.

Single control-plane instance Each Ankaaios cluster runs exactly one server instance. As a consequence, the control plane is not replicated by default, which makes the server a potential single point of failure.

Limited built-in service management Ankaaios does not provide higher-level cloud-native features such as service discovery, load balancing, or an integrated networking abstraction. These concerns must be handled separately by the surrounding software stack.

External handling of inter-workload communication Because communication between workloads is out of scope for the orchestrator, application developers or system integrators must provide the required communication mechanisms explicitly. This keeps Ankaaios lightweight, but it also shifts part of the integration effort to the application or platform level.

Control Interface availability depends on the runtime The Control Interface is currently available for workloads using the `podman` runtime, but not for those using `podman-kube`. This is not a fundamental limitation of the architecture, but it is a practical limitation of the current implementation.

Early-stage ecosystem Compared with more mature orchestration platforms, Ankaaios still has a smaller ecosystem of tools, integrations, and community resources. As a result, adopting it in complex projects may require more project-specific integration work.

4.5 Comparative evaluation

This section presents a systematic comparison of three container orchestration platforms — **K3s**, **Eclipse Ankaio**s, and **Eclipse BlueChi** — evaluating their suitability for automotive edge computing. The assessment focuses on **startup performance**, **memory footprint**, and **functional capabilities**, providing insights into how each platform handles a standardized workload under constrained resources.

A dedicated test bench was established to ensure a fair and reproducible evaluation. Each platform was deployed in isolation on an identical virtual machine, provisioned with **Vagrant** and configured with **Ansible**, using the same reference application: an **nginx** web server. The benchmark intentionally uses a minimal single-node deployment for each platform in order to compare baseline footprint and lifecycle overhead; this does not imply that the platforms themselves are limited to single-node operation.

4.5.1 Evaluation methodology and test environment setup

Each orchestrator is installed in a separate VM with identical specifications. The environment is fully managed via Vagrant for provisioning and Ansible for automated configuration, ensuring **consistency** and **reproducibility**.

Virtual Machine Specifications.

All experiments were executed on a set of virtual machines running *Ubuntu 22.04 LTS* (Jammy) as the operating system. Each virtual machine was configured with *2 virtual CPUs* and *2048 MB of RAM* and was provisioned using *VirtualBox* as a virtualization provider.

To ensure isolation between the evaluated orchestration platforms, a dedicated virtual machine was used for each orchestrator under analysis, namely **bluechi-vm**, **ankaio-vm**, and **k3s-vm**.

This virtual machine configuration was used only for the comparative evaluation presented in Chapter 4. The experimental campaign discussed in Chapter 6 was conducted in a separate virtualized environment with different resource allocation, tailored to the Atlas benchmark.

All machines were initialized from a common baseline configuration, defined in Appendix A Listing 3. Before deploying the orchestrator-specific components, each VM executed a shared Ansible role responsible for installing the required base packages, including Docker, Podman, **psutil**, **pandas**, and **sysstat**. This ensured a consistent environment across all experiments prior to applying the orchestrator-specific roles.

Orchestrator deployment

Each platform is installed and configured according to its official procedure:

Orchestrator	Deployment & Workload Management
K3s	Installed via official script; deploys nginx as a Kubernetes Deployment + Service (NodePort)
Ankaaios	Installed via official script; uses Podman as runtime; nginx defined declaratively in <code>state.yaml</code>
BlueChi	Installed from PPA; controller + agent run on the same node; nginx managed through a systemd unit that starts a Podman container and is controlled via <code>bluechictl</code>

Table 4.1: Orchestrator deployment strategies

Although all orchestrators run the same reference workload (`nginx` container, single replica), the **management mechanism differs**, which is the key focus of this comparative evaluation.

Monitoring and metrics collection

A custom monitoring framework was used to collect detailed operational metrics:

- **Monitor script:** `monitor_activity.py` (Python + `psutil`)
 - Configurable sampling intervals (default 1s/5s)
 - Per-process metrics: CPU usage (%), RSS (MB), VMS (MB), I/O bytes (read/write)
 - Data stored in CSV for later analysis
- **Analysis script:** `analyze_footprint.py` computes mean and peak values per process
- **Workload exercise:** orchestrator-specific scripts perform comparable lifecycle operations:
 1. Burst of HTTP requests to `nginx`
 2. Start and stop of the managed workload
 3. Restart or redeployment of the managed workload, where supported

Aspect	Detail
VM OS	Ubuntu 22.04 LTS
CPU	2 vCPUs
RAM	2048 MB
VM provisioning	Vagrant Virtual Box
Configuration management	Ansible (role based)
Monitoring tool	Python psutil (1s sampling)
Reference workload	Nginx container (1 replica)
Workload operations	HTTP requests, start/stop, restart or redeployment

Table 4.2: Summary of evaluation methodology

4.5.2 Resource Consumption Analysis

To quantify the **runtime overhead** introduced by each orchestrator, the measurements were collected after the nginx container was fully deployed and operational, considering idle conditions. This baseline measurement reflects conditions similar to those of automotive ECUs, where the orchestrator’s steady-state overhead must remain sufficiently low to allow coexistence with other, potentially safety-critical, functions that extend beyond its own responsibilities.

Each orchestrator was monitored for approximately 45 seconds at a 1-second sampling interval using the *monitor_activity.py* script. The collected per-process time series were then aggregated by the *analyze_footprint.py* script, which computes per-process and total average values for CPU usage (%) and Resident Set Size memory (RSS, in MB)

Per-process footprint

This data reports the total orchestrator footprint obtained by summing the contributions of all constituent processes at each sampling instant and then computing the overall mean and maximum values.

Orchestrator	Process	Avg CPU (%)	Max CPU (%)
K3s	k3s-server	24.51	46.90
Ankaaios	ank-server	0.03	1.00
	ank-agent	0.21	1.00
BlueChi	bluechi-controller	0.11	2.00
	bluechi-agent	0.00	0.00

Table 4.3: Per-process CPU usage at idle with deployed workload.

K3s exhibits the highest overhead by a significant margin, consuming on average **551.68 MB** of **RSS** and **24.51% CPU**. This is expected, as K3s packages the Kubernetes control-plane logic together with an embedded datastore and bundled cluster

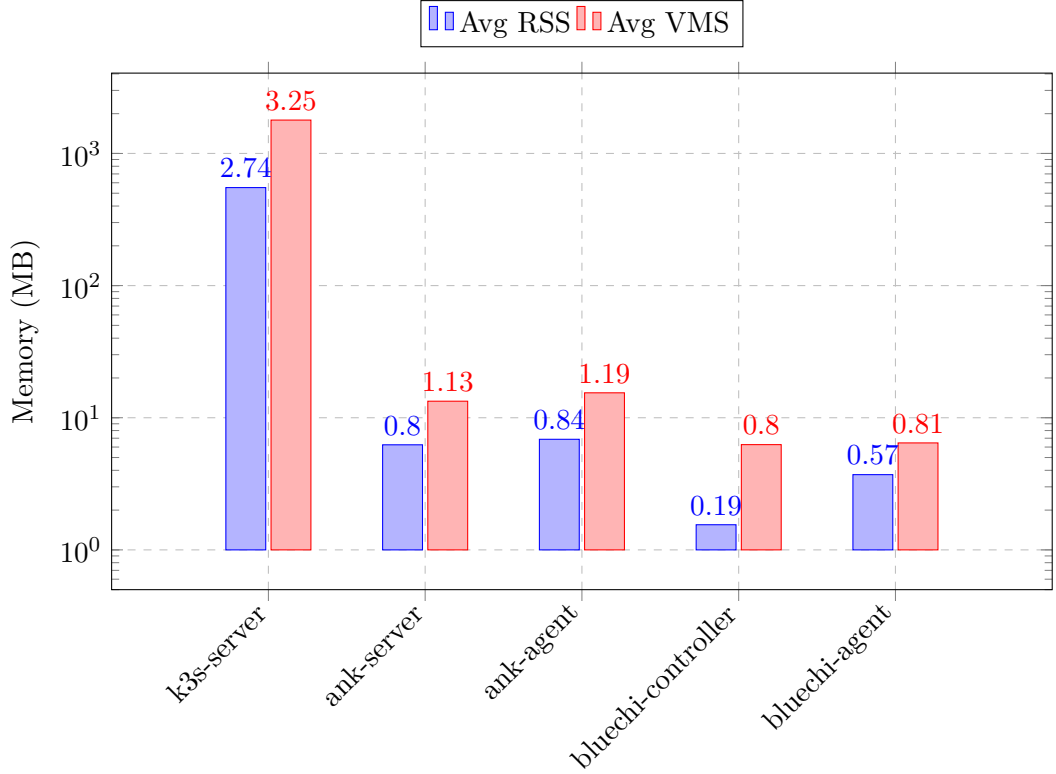


Figure 4.6: Memory usage per process for different orchestrators (logarithmic scale)

services in a lightweight distribution. While this **design simplifies deployment**, it results in a memory footprint that is approximately $42\times$ larger than Ankaaios and $105\times$ larger than BlueChi.

Ankaaios maintains a remarkably **lean profile**, with a combined RSS of only 13.11 MB and near-zero CPU utilization (0.23% average). The agent process is slightly heavier than the server (6.87 vs. 6.24 MB RSS), which is consistent with its role of interfacing with the Podman runtime used in this benchmark. The maximum CPU spike of 1.00% is compatible with periodic reconciliation or runtime-management activity, confirming that the platform is well-suited for environments where CPU cycles must remain available for timing-sensitive vehicle functions.

BlueChi achieves the **smallest footprint** of the three, with a total RSS of only 5.27 MB and virtually no CPU consumption at steady state. This is attributable to its architectural decision to delegate all container lifecycle management to systemd, meaning the BlueChi processes themselves serve only as a thin coordination layer. The controller consumes just 1.55 MB, and the agent 3.71 MB, with no observable variation between average and maximum values — indicating an **extremely stable** and predictable **memory profile**.

The following ratio summarizes the relative memory cost of each solution, normalized to BlueChi as the baseline:

	BlueChi	Ankaaios	K3s
RSS Ratio	1×	2.5×	104.7×

Table 4.4: Relative memory cost normalized to BlueChi as baseline.

4.5.3 Functional Capabilities

Beyond footprint measurements, the functional capabilities of the candidate orchestrators are also relevant to the selection.

Table 4.5 provides a comparative overview of the functional capabilities offered by the three orchestrators under evaluation.

K3s emerges as the most **feature-rich** platform, inheriting the breadth of the Kubernetes ecosystem. It supports declarative workload definitions through standard **YAML manifests**, built-in service discovery and load balancing, secrets management, and fine-grained resource limits via Kubernetes and cgroups. Its **Kubernetes API** compatibility also enables seamless integration with a **vast ecosystem** of third-party tools and extensions. However, these capabilities come with the largest control-plane footprint among the three candidates.

Ankaaios offers a deliberately reduced but automotive-oriented feature set. It supports declarative workload management through a **YAML-based** desired-state model, provides **state-based health** supervision, and manages **OCI container images** through the selected runtime. In the implementation used in this thesis, the runtime is Podman, which is also the basis for the Atlas prototype in the following chapter. At the same time, Ankaaios does not provide built-in service discovery, load balancing, or secrets management.

Feature	K3s	Ankaaios	BlueChi
Managed workload definition	Kubernetes YAML	YAML manifest	systemd units
Built-in rolling update strategy	YES	NO	NO
Health monitoring	YES	YES (state-based)	Via systemd
OCI image support	YES	YES	Indirect (via Podman)
Service discovery & networking	YES	NO	NO
Volumes	YES	YES (runtime-managed)	Indirect (via systemd/Podman)
Primary execution model	Pods	Workloads	Systemd Services
Multi-node support	YES	YES	YES
Kubernetes API compatibility	YES	NO	NO
Secrets management	YES	NO	NO
Built-in load balancing	YES (ServiceLB)	NO	NO
Remote lifecycle control	YES (kubect1)	YES (ank CLI / API)	YES (bluechict1 / D-Bus)
Automotive / embedded focus	NO	YES	YES
Resource limits & quotas	YES (Kubernetes)	YES (runtime cgroups)	Basic (systemd cgroups)
Logging & observability	YES	Basic (CLI + journal)	Basic (journal)

Table 4.5: Functional capabilities comparison across the three orchestrators.

BlueChi occupies a unique position as a distributed **systemd service controller**. It does not define its own container abstraction and therefore manages containerized

workloads indirectly, for example through Podman-backed systemd units or quadlets. This design yields the smallest orchestration overhead, but it also means that advanced container-oriented features such as rolling updates, declarative reconfiguration, and container-aware health policies must be implemented through surrounding tooling.

Both Ankaaios and BlueChi are explicitly designed for automotive or embedded environments, whereas K3s targets general-purpose edge and cloud deployments. Resource limiting is available in all three, although with different levels of granularity — from Kubernetes-native controls in K3s to runtime- or systemd-level cgroup controls in Ankaaios and BlueChi.

4.5.4 Discussion

The evaluation highlights three clearly distinct design approaches to container orchestration in constrained environments. While all three platforms are capable of managing containerized workloads, their architectural priorities lead to significantly different trade-offs in terms of resource usage and operational capabilities.

From a purely resource-oriented perspective, **BlueChi** demonstrates the smallest footprint. Its design leverages the native *systemd* service manager to execute workloads, which minimizes orchestration overhead and results in a very stable and predictable runtime profile. This approach is particularly attractive for extremely resource-constrained systems, but it also shifts several orchestration responsibilities outside of the platform itself.

At the opposite end of the spectrum, **K3s** provides a comprehensive orchestration environment derived from Kubernetes. This architecture enables a wide range of advanced features and integrations but introduces a significantly larger control plane, which translates into a higher CPU and memory overhead. Although K3s is designed as a lightweight Kubernetes distribution, its architecture remains oriented toward cloud-native deployments rather than embedded automotive platforms.

Ankaaios occupies an intermediate position between these two extremes. Its architecture provides native support for containerized workloads through a declarative desired-state model while maintaining a relatively small resource footprint. By avoiding cloud-oriented features that are unnecessary in in-vehicle systems, the platform achieves a balance between orchestration functionality and operational efficiency.

These results suggest that solutions specifically designed for embedded and automotive environments tend to adopt simpler orchestration models compared to general-purpose cloud platforms. In such contexts, the ability to minimize infrastructure overhead while still supporting reliable workload lifecycle management becomes a key design objective.

4.6 Final selection and rationale

Based on the results of the comparative evaluation, **Eclipse Ankaios** was selected as the orchestration platform for this project.

The analysis highlighted three distinct design philosophies among the evaluated solutions. **K3s** provides the richest feature set, inheriting the full Kubernetes ecosystem, but at the cost of a significantly higher resource footprint. As demonstrated in the resource consumption analysis, its memory and CPU overhead are considerably larger than those of the other candidates, making it less suitable for resource-constrained automotive ECUs.

At the opposite end of the spectrum, **Eclipse BlueChi** achieves the smallest runtime footprint by delegating workload execution to *systemd*. While this architecture results in an extremely lightweight and stable orchestration layer, it also limits the feature set available out of the box. Containerized workloads can still be managed indirectly through *systemd* units and Podman integration, but higher-level container orchestration semantics require additional external mechanisms.

Eclipse Ankaios represents a balanced compromise between these two approaches. It maintains a **very small resource footprint** comparable to lightweight embedded solutions while still providing a **container-oriented orchestration model** with declarative workload management, health supervision, multi-node support, and runtime-backed execution of OCI images. In the implementation used in this work, these capabilities are exercised through the Podman runtime and a YAML startup manifest, which makes Ankaios a natural fit for the prototype developed in the following chapters.

For these reasons, Ankaios offers the most appropriate trade-off between **lightweight resource usage** and **functional capabilities**. It provides sufficient orchestration primitives to manage containerized workloads while preserving architectural simplicity and leaving room for domain-specific extensions when required.

Consequently, **Ankaios was selected as the orchestration foundation** for the system architecture implementation presented in the following chapters.

Chapter 5

Implementation

Following the comparative evaluation presented in the previous chapter, **Eclipse Ankaaios** was selected as the orchestration platform for the proposed system architecture. Its lightweight design and its focus on automotive and edge environments make it a suitable foundation for managing containerized workloads on in-vehicle computing platforms.

This chapter describes the practical implementation built on top of Ankaaios. The orchestrator provides the core mechanisms for workload lifecycle management and declarative deployment, but the prototype also requires domain-specific management functions that are outside the scope of the base orchestrator, such as coordinated dependency handling, controlled workload relocation, runtime network reconfiguration for distributed KUKSA components, and experiment-oriented logging.

To support these requirements, an additional management framework named **Atlas** was developed using the **Ankaaios Python SDK** [14]. Atlas does not replace the orchestrator; rather, it acts as a workload-managed control application that exposes a higher-level command-line interface and implements the orchestration support logic needed by the prototype.

The following sections present the architecture and implementation details of Atlas, including how it integrates with Ankaaios, how workloads are described and managed, and how benchmarking data is collected for the experimental evaluation.

5.1 System architecture

To address the limitations identified in the previous chapter, a domain-specific management framework named **Atlas** was designed and implemented on top of Ankaaios. Atlas provides an additional management layer for containerized workloads interacting with the KUKSA data ecosystem, enabling a more structured operational model for the prototype.

Atlas is itself deployed as a **regular workload managed by Ankaaios**. In the default configuration it runs on the main node of the test environment, but it remains subject to the same placement and lifecycle rules as any other managed workload. This choice keeps the design consistent with the orchestration model under

study: Atlas uses the services of the orchestrator while remaining external to the orchestrator implementation.

From the operator’s perspective, Atlas exposes a simplified command-line interface for starting, stopping, moving, updating, loading, and inspecting workloads without requiring direct interaction with low-level control-plane primitives. Internally, Atlas is implemented using the Ankaaios Python SDK and accesses the control interface made available to managed workloads. In this way, it mediates between user-level management commands and the desired-state mechanisms of the orchestrator, while leaving the core orchestration responsibilities to Ankaaios itself.

This architectural role is intentionally narrow. Atlas does not replace the orchestrator and it does not introduce a second orchestration plane. Instead, it complements Ankaaios with domain-specific logic for dependency handling, configuration propagation, controlled workload relocation, and coordinated management of the KUKSA-related workloads used in the prototype.

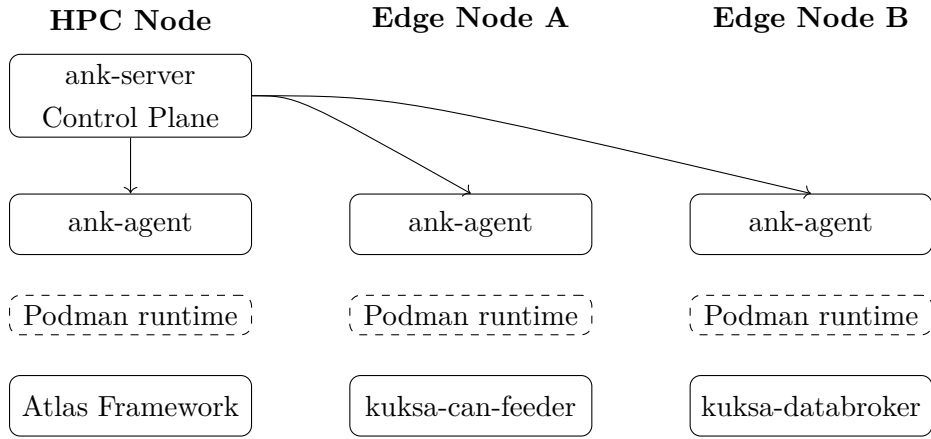


Figure 5.1: Deployment architecture of the Atlas framework within the Ankaaios cluster. The HPC node hosts the Ankaaios control plane and the Atlas management layer, while edge nodes run ank-agent instances responsible for managing containerized workloads through the Podman runtime.

5.1.1 Deployment and bootstrap

Atlas is packaged as a container image and deployed as a standard Ankaaios workload. Its entry point is `atlas.py`, which connects to the Ankaaios control interface through the SDK and provides the command-line interface used to manage workloads at runtime.

From an architectural perspective, an important design requirement was to ensure that the Atlas process remains active even if the interactive terminal session is interrupted. This allows the container to continue running under Ankaaios’ supervision and to be reattached later without affecting the managed workloads.

Deployment is bootstrapped through a `state.yaml` file placed in `/etc/ankaaios/state.yaml` before Ankaaios is started. This file declares Atlas in the desired state, specifies the container image and runtime options, and includes a `controlInterfaceAccess` block

granting `ReadWrite` permissions on the relevant state objects. These permissions are required for Atlas to query and modify the cluster state through the SDK.

The runtime configuration also defines four bind mounts:

- `/run/podman/podman.sock`, to access the Podman API and collect runtime information about containers;
- `/etc/ankaaios/kuksa-feeder-files`, to make the CAN feeder configuration files available inside the container;
- `/etc/ankaaios/logs`, mounted at `/app/logs`, to store the `ankaaios_operations.log` file;
- `/etc/ankaaios/workloads`, to make custom workload YAML files available to Atlas, as described in Section 5.2.2.

These host paths must be created before Ankaaios is started, otherwise the corresponding bind mounts cannot be established by Podman.

5.1.2 Container runtime: Podman

In the implemented architecture, *Podman* has been selected as the container runtime responsible for executing application workloads on each node of the Ankaaios cluster. The choice is motivated by several architectural and security advantages that make it particularly suitable for embedded and automotive-oriented environments.

One of the main characteristics of Podman is its *daemonless architecture*. Unlike container engines such as Docker, which rely on a centralized daemon process to manage container lifecycle operations, Podman follows a fork-exec model in which containers are launched directly as child processes of the invoking user. This design removes the need for a long-running privileged background service and eliminates a potential single point of failure in the container management layer.

Podman supports **rootless execution** and can **reduce daemon-related overhead**. Containers can be started by non-privileged users without requiring system-level access to the host. In practice, even if a process inside a container runs with root privileges, it is mapped to an unprivileged user on the host through Linux user namespaces, significantly reducing the risk of privilege escalation and limiting the potential impact of container escapes or compromised workloads.

Another important aspect is Podman's tight integration with *systemd*. In the absence of a dedicated daemon, Podman can rely on *systemd* units to manage container lifecycle operations such as startup, shutdown, restart policies, and logging, allowing containers to be treated as native system services.

From a resource management perspective, the absence of a continuously running daemon also contributes to reduced runtime overhead. Since no background service needs to remain active when containers are not running, system resources are only consumed when containers are actually executed. This lightweight behaviour is particularly advantageous in automotive High Performance Computing (HPC) platforms where computational resources must be carefully controlled.

Finally, Podman is fully compliant with the *Open Container Initiative (OCI)* standards, guaranteeing compatibility with container images built for other OCI-compliant runtimes, including Docker. Existing container images can therefore be reused across OCI-compliant runtimes on the same target architecture, improving portability and interoperability across container ecosystems.

5.1.3 Container networking

In containerized environments, the networking model determines how containers communicate with the host system, with each other, and with external services. By default, Podman can create bridge-based networks in which each container is connected to a virtual Ethernet bridge on the host and reaches external services through network address translation and port-forwarding rules. Although this mode provides isolation, it introduces an additional networking layer that is unnecessary for the prototype considered here.

The implemented architecture adopts *host networking*. When a container is started with the `--network host` option, it shares the host network namespace directly and uses the host's IP addresses, routing table, and port bindings without an intermediate bridge. This choice reduces networking indirection, simplifies the configuration of the KUKSA components, and matches the relatively static topology of the evaluation setup.

Ankaaios does not manage inter-workload communication or service discovery. It forwards runtime options, including the `--network` flag to the selected container runtime, while actual connectivity remains determined by the underlying host network and by the options declared in the workload YAML files. As a consequence, when a workload is moved to another node, Ankaaios handles the redeployment but does not automatically update the connection parameters of dependent services.

This behaviour is particularly relevant for the KUKSA Databroker, whose clients must know the correct endpoint after relocation. The corresponding reconfiguration is handled explicitly by the Atlas network configuration subsystem described in Section 5.1.4. Since host-networked containers share the host port space, the workload definitions also need to avoid port conflicts by construction.

5.1.4 Network configuration management

A practical consequence of the chosen networking model is that workloads must be told explicitly how to reach services that may move across nodes. In the prototype, this issue mainly concerns the communication between the *KUKSA Databroker* and its client workloads, namely *kuksa-databroker-cli* and *kuksa-can-feeder*. When all three workloads run on the same node, `localhost:55555` is sufficient. When the Databroker is moved to another node, the client-side endpoint must be updated accordingly.

To address this requirement, Atlas implements a small network-configuration subsystem built around a mutable in-memory registry containing the current Databroker

host and port. These values are initialized with the default endpoint `localhost:55555` and can be modified at runtime through the `config` command.

The configuration workflow follows a two-phase approach composed of **set** and **apply**. During the `config set` phase, the operator updates the desired host and/or port values without affecting any running workload. The changes only become effective when `config apply` is executed.

The `config apply` command propagates the new settings through three main steps:

1. **Configuration update.** Atlas regenerates `kuksa-databroker-cli.yaml` so that the CLI container receives the updated endpoint through environment variables. It also restores `kuksa-can-feeder.yaml` to its base form and updates the host-mounted `dbc_feeder.ini` file, which is the actual source of the feeder connection parameters.
2. **Runtime inspection.** Atlas queries the current state through the SDK to determine which dependent workloads are currently running and on which agent they are deployed.
3. **Controlled restart.** If the affected workloads are running, Atlas offers to stop and redeploy them using the updated configuration while keeping each workload on its current agent.

This mechanism does not implement automatic service discovery; instead, it provides an explicit and predictable reconfiguration path that is consistent with the lightweight nature of the prototype. Additional helper commands such as `config show` and `config reset` allow the operator to inspect the current values and restore the default endpoint when the system returns to a single-node configuration.

5.1.5 KUKSA vehicle abstraction layer

Eclipse KUKSA provides the runtime infrastructure used in the prototype to expose vehicle data through the VSS model introduced in Chapter 3. In this implementation, **KUKSA Databroker** acts as the central signal broker: it stores the current value of the available VSS signals and exposes them primarily through a gRPC interface. The same signal model remains visible to client applications independently of the native CAN source from which the data originates.

The data pipeline used in the prototype is composed of three cooperating workloads:

- **kuksa-databroker**—the signal broker service that stores and serves VSS data.
- **kuksa-can-feeder**—a feeder process that replays raw CAN frames from a `candump` log, decodes them through a `.dbc` file, maps the decoded values onto VSS paths, and publishes the resulting updates to the Databroker.

- **kuksa-databroker-cli**—an interactive client used to inspect and manipulate individual signals during test and validation activities.

This arrangement makes it possible to exercise the full signal path without depending on a physical vehicle during development. The `candump` trace emulates a live CAN stream, the `.dbc` file provides the semantic interpretation of the frames, and the Databroker exposes the resulting values in a normalized form to the rest of the software stack.

A relevant consequence of this arrangement is that the VSS signal namespace remains stable even if the Databroker is relocated to another node. What changes is the communication endpoint, not the logical signal paths consumed by applications. For this reason, workload relocation and network reconfiguration are treated as separate concerns: the `move` command changes placement, while `config apply` updates the Databroker endpoint used by the feeder and the CLI.

5.2 Workloads

5.2.1 Definition and configuration model

Each workload managed by Atlas is described by a *YAML configuration file* that specifies the container runtime parameters required by Podman. These files follow a minimal schema composed of three fields:

- **image**: the fully qualified container image reference, including the registry, repository, and tag (e.g., `docker.io/oxhunt/kc-provider:latest`).
- **commandOptions**: a JSON array of command-line flags passed to the container runtime at launch time, including options such as `-network host` for host networking, `-v` for bind-mounting host directories, and `-env` for injecting environment variables.
- **commandArgs**: a JSON array of arguments passed to the container's entrypoint process, used to configure the application running inside the container.

As an example, the *KUKSA CAN Feeder* workload is defined as follows:

```

1 image: docker.io/oxhunt/kc-provider:latest
2 commandOptions: [ "--network", "host", "-v",
3   "/etc/ankaaios/kuksa-feeder-files:/kuksa-feeder-files" ]
4 commandArgs: [ "--config", "/kuksa-feeder-files/dbc_feeder.ini" ]

```

Listing 1: KUKSA CAN Feeder YAML configuration file

These YAML files serve as the single source of truth for the runtime configuration of each workload. The built-in workload definitions are stored inside the Atlas

container, while additional workload files can be mounted from the host and loaded dynamically at runtime.

At the application level, the set of available workloads and their inter-workload relationships are defined in a centralized *configuration registry*, implemented as a Python dictionary (`AVAILABLE_WORKLOADS`). Each entry maps a workload name to two properties: the path to its YAML configuration file and a list of *declared dependencies*. Dependencies are expressed as pairs of a workload name and a condition type. The condition `ADD_COND_RUNNING` specifies that the dependency must have reached a running state before the dependent workload is scheduled. For instance, both the *Databroker CLI* and the *CAN Feeder* declare a dependency on the *KUKSA Databroker* with this condition.

When a workload is started, Atlas uses the `WorkloadBuilder` class provided by the SDK to construct a workload specification programmatically, configured with the workload name, the target agent, the runtime type (`podman`), the restart policy, and the runtime configuration loaded from the corresponding YAML file. For each declared dependency, the `add_dependency()` method is invoked on the builder, injecting the constraint into the specification. The resulting object is then submitted to the control plane through `apply_workload()`, which registers the workload in the desired cluster state.

The following listing illustrates the core logic of this process, showing how the workload specification is constructed, how dependencies are injected, and how the result is submitted to the control plane.

This design separates the *runtime configuration* (defined in YAML files and modifiable at runtime) from the *structural configuration* (defined in the Python registry and describing workload relationships), enabling independent evolution of deployment parameters and dependency topology.

5.2.2 Lifecycle operations

Atlas exposes five core operations for managing the lifecycle of containerized workloads: *start*, *stop*, *move*, *update*, and *load*. A separate *status* command is provided for runtime inspection of the cluster state. All lifecycle operations interact with the control plane through the Python SDK and record structured execution data for subsequent analysis.

Start

The start operation deploys a new workload on a specified agent. The operator provides the workload name and optionally the target agent and restart policy; if omitted, these default to `agent_A` and `NEVER`, respectively.

The operation constructs a workload specification using the `WorkloadBuilder` class as described in the previous section. If the workload declares dependencies, each constraint is injected through `add_dependency()`, ensuring that Ankaio defers activation until the required upstream services are available. The resulting specifica-

tion is then submitted via `apply_workload()`, which updates the desired state. An example of this operation is provided in Appendix A Listing 4.

Atlas records two distinct timing measurements during this process. The first is the duration of the SDK request itself, logged as a **START** entry. The second is the time required for the workload to actually reach the **RUNNING** state, measured by polling the cluster state through `wait_for_running()` and recorded as a separate **STARTUP** entry. After a successful startup, Atlas also attempts to collect the workload's memory usage through the mounted Podman socket and records it as a **MEMORY** entry.

Stop

The stop operation removes a running workload from the cluster. Before issuing the deletion command, Atlas performs a *runtime dependency analysis*: it queries the current cluster state and inspects the dependency declarations of all known workloads to identify any running workload that depends on the one being stopped.

If dependent workloads are found, Atlas stops them first in a *cascading fashion*, issuing individual `delete_workload()` calls for each dependent before deleting the target workload itself. This prevents dependent containers from entering an error state due to a missing upstream service. Each individual stop operation is logged independently, providing granular visibility into the cascading sequence.

Status

The status operation provides a real-time view of the cluster state. Atlas queries the control plane via `get_workload_states()` and presents the results in a tabular format, grouping workloads by agent. For each workload instance, the output includes the workload name, its execution state (e.g., **RUNNING**, **STOPPING**), and the sub-state that provides additional detail on the current condition (e.g., **OK**, **LOST**). The list of connected agents is also reported, giving the operator a complete picture of the cluster topology at a glance.

Move

The move operation redeploys across nodes a running workload, exploiting a brief transition period. Atlas first validates that the workload is running and that the target agent is available by querying the cluster state.

```

1 # Retrieve the current workload specification
2 result = ankaios.get_workload(name_workload)
3 workload = result[0] if isinstance(result, list) else result
4
5 # Reassign the target agent in-place
6 workload.update_agent_name(target_agent)
7
8 # Submit the updated specification to the Ankaaios control plane
9 ankaios.apply_workload(workload)

```

Listing 2: Workload relocation across nodes through the Ankaaios SDK

The relocation is implemented through a three-step sequence: the current workload specification is retrieved via `get_workload()`, the target agent is updated using `update_agent_name()`, and the modified specification is resubmitted through `apply_workload()`. Ankaaios then applies the updated specification by stopping the container on the original agent and redeploying it on the target one. This process introduces a brief transition period during which the workload is temporarily halted on both nodes.

It is important to note that the move operation only handles the redeployment of the container. If other workloads communicate with the migrated service via its IP address, the network configuration must be updated separately using the `config` subsystem described in Section 5.1.4.

Load

The load operation extends the system’s flexibility by allowing the operator to register and deploy a *custom workload* at runtime, without modifying the built-in configuration registry or rebuilding the Atlas container image.

The operator provides a workload name, a YAML configuration file, and optionally a target agent and dependency declarations. The YAML file follows the same schema used by the pre-configured workloads (`image`, `commandOptions`, `commandArgs`).

To make custom YAML files accessible inside the Atlas container, a dedicated bind-mount is configured in the state file: the host directory `/etc/ankaaios/workloads/` is mounted into the container at the same path. When the operator places a YAML file on the host, it becomes immediately visible inside the container without requiring a restart or image rebuild. If only a filename is provided rather than an absolute path, Atlas resolves it automatically relative to this mounted directory.

Upon invocation, the load command performs two actions. First, it registers the new workload in the in-memory `AVAILABLE_WORKLOADS` dictionary, making it available to all other Atlas operations (`start`, `stop`, `move`, `update`) for the lifetime of the running Atlas process. Second, it immediately deploys the workload on the specified agent using the same `WorkloadBuilder` pipeline, including the injection of any declared dependencies.

This mechanism enables a *plugin-like extensibility model*: new containerized

services can be added to the managed environment without modifying the Atlas source code or its container image, simply by placing a YAML file on the host filesystem.

A reference implementation of the load operation is provided in Appendix A, Listing 5

Update

The update operation replaces the container image of a workload with a different version. The operator specifies the workload name and optionally a new image tag; if omitted, the tag defaults to `latest`.

The operation begins by modifying the workload’s YAML configuration file: the `image` field is parsed to extract the base image reference, and the tag portion is replaced with the requested value. If the workload is not currently running, the configuration change is persisted and will take effect on the next start.

If the workload is running, Atlas performs a *delete-and-redeploy* sequence: the current instance is removed via `delete_workload()`, followed by a brief stabilisation pause to allow the container runtime to release resources, and then a new instance is deployed using the updated configuration file through the standard `WorkloadBuilder` pipeline. The workload is restarted on the same agent where it was previously running, preserving the original placement. This approach results in a brief period of *planned downtime* during the transition, as the current Ankaio SDK does not expose a native in-place image update mechanism.

Once the new instance has been deployed, Atlas verifies that it reaches the `RUNNING` state within 60 seconds using the same `wait_for_running()` polling mechanism described in Section 5.2.2. If the updated workload starts successfully, the operation is logged as a success.

If, however, the new image fails to start—for example because the tag does not exist in the registry, the image is incompatible with the target architecture, or the application crashes at launch—Atlas triggers an *automatic rollback*. The Ankaio SDK does not provide a native rollback primitive; therefore, this mechanism is implemented entirely within Atlas and proceeds as follows:

1. The failed workload instance is removed.
2. The YAML configuration file is rewritten with the *previous* image reference using the `rewrite_image_tag()` utility.
3. A new instance is deployed from the restored configuration on the same agent.
4. `wait_for_running()` is invoked again to confirm that the reverted workload reaches the `RUNNING` state.

The outcome of the rollback is recorded in the operations log with a dedicated `ROLLBACK` entry, whose status is `SUCCESS` if the original image was restored successfully, or `FAILED` if the revert also did not reach a healthy state. This guarantees that,

in the common failure scenario of a bad image tag, the system self-heals without operator intervention, preserving workload availability at the cost of an extended downtime window (up to two full startup cycles rather than one).

5.2.3 Dependency Management

In the KUKSA data ecosystem, certain workloads require other services to be operational before they can function correctly. Specifically, the *Databroker CLI* and the *CAN Feeder* both require the *KUKSA Databroker* to be running, as they connect to it at startup to exchange vehicle signal data.

Atlas manages these relationships through a *declarative dependency model*. Dependencies are statically defined in the configuration registry, where each workload entry may include a list of dependencies expressed as pairs of a workload name and a condition. The condition `ADD_COND_RUNNING` specifies that the dependency must have reached a running execution state before the dependent workload is scheduled for deployment.

At *start time*, dependencies are injected into the workload specification through the `WorkloadBuilder` API. Ankaaios uses this information to *defer the scheduling* of the dependent workload until all its dependencies are satisfied. This ensures that the CAN Feeder, for example, is not started before the Databroker is available, preventing connection failures during the container initialization phase.

At *stop time*, the dependency model operates in the opposite direction. When the operator requests the deletion of a workload, Atlas dynamically identifies all currently running containers that declare a dependency on the target. These dependent workloads are stopped first, following a *dependents-first* ordering, before the target is removed. This cascading behaviour prevents situations where a client workload continues to run without its required upstream service, which could lead to persistent error states or uncontrolled restart loops.

This two-phase dependency enforcement—startup ordering managed by Ankaaios and shutdown cascading managed by Atlas—provides a consistent operational model that respects inter-workload relationships throughout the workload lifecycle.

5.3 Operational logging

Every relevant action executed by Atlas is recorded in a structured log file named `ankaaios_operations.log`. Inside the container, the file is written under `/app/logs`; through the bind-mount defined in `state.yaml`, it is persisted on the host under `/etc/ankaaios/logs`.

Operational entries are produced by the `log_operation()` utility and follow a regular key–value structure, for example:

```
1 [2026-03-12 14:15:51] UPDATE | Workload: nginx |
2 Duration: 21.344s | Status: SUCCESS | Error: docker.io/library/nginx
   :1.25.0 -> docker.io/library/nginx:latest
```

The log contains multiple entry types. **START**, **STOP**, **MOVE**, and **UPDATE** record the direct execution time of the corresponding management command; **STARTUP** records the time needed for a workload to actually reach the **RUNNING** state; **ROLLBACK** records the outcome of a recovery action after a failed update; and **MEMORY** records the measured memory usage of a successfully started workload. Each entry includes a timestamp, the workload name, a status field, and, when applicable, an additional diagnostic or informational message in the **Error** field.

The file is opened in append mode, so entries accumulate across multiple runs instead of overwriting previous observations. This persistent log is the basis for the benchmarking and post-processing activities described in the next section.

5.4 Benchmarking and analysis tooling

While the previous sections described the operational mechanisms implemented in Atlas, evaluating the effectiveness and performance of these mechanisms requires a systematic measurement methodology. For this purpose, a set of benchmarking and analysis tools was developed to collect and analyze execution metrics during workload lifecycle operations.

To generate a statistically meaningful dataset for the experimental evaluation, two companion tools were developed alongside Atlas: `benchmark.py` and `log_analyzer.py`.

5.4.1 Benchmark driver

`benchmark.py` is an automated driver that exercises the Atlas operations across the configured workloads and agents, repeating the full cycle for a configurable number of rounds. It is executed inside the Atlas container and writes to the same log file used by the interactive CLI through the shared logging utilities.

Within each round, the script first orders the workloads so that independent services are handled before dependent ones. It then performs the following sequence:

1. **START** each workload on the primary agent.
2. **MOVE** each running workload to the secondary agent, if a second agent is available.
3. **UPDATE** the image tag of each workload to a benchmark-specific tag.
4. **STOP** all workloads in reverse order, so that dependents are removed before the services they rely on.
5. If a secondary agent is available, execute an additional **START-STOP** cycle for the independent workloads on that agent to collect more placement data.

During each start phase, the benchmark also measures startup latency and attempts to record memory consumption through the Podman API, using the same

helper functions employed by the interactive CLI. A configurable pause between operations (default: 2 s) allows the runtime to settle between successive actions. After each round, the original image tags are restored so that the following round starts from a clean baseline.

The benchmark accepts two command-line arguments: `-rounds`, which controls the number of full cycles, and `-pause`, which controls the delay between operations.

5.4.2 Log analyser

`log_analyzer.py` is a standalone post-processing script that parses the accumulated log file and produces both a textual report and a set of charts. It can process both duration-oriented lifecycle entries and `MEMORY` entries recorded after successful starts.

The textual report presents aggregated statistics—count, mean, minimum, and maximum—at multiple levels of granularity: per operation type, per workload, and per workload–operation pair. A separate section summarizes memory-usage statistics per workload, while the global outcome distribution is used to compute the success/failure ratio.

The current implementation generates and saves the following charts as PNG files:

- a box plot of operation durations grouped by operation type;
- a grouped bar chart of average duration per workload and operation, with min/max error bars;
- a pie chart of the overall success/failure ratio;
- a timeline scatter plot showing how operation durations evolve over time;
- a comparison chart between `START` request time and `STARTUP` time-to-RUNNING;
- a memory-usage bar chart showing the average memory consumption per workload, again with min/max error bars.

Together, `benchmark.py` and `log_analyzer.py` provide the measurement pipeline used to generate the results discussed in the following chapter.

5.5 Summary

This chapter presented the implementation of the Atlas management framework built on top of Eclipse Ankaio. The architecture of the framework, its deployment model, the workload description format, the lifecycle operations, the dependency model, and the configuration mechanisms required by the KUKSA-based signal pipeline were described in detail.

The chapter also introduced the structured logging and benchmarking utilities used to evaluate the prototype. By combining command-level timing, startup-time

measurement, memory sampling through the Podman API, and automated benchmark execution, the implementation provides a reproducible basis for the experimental analysis presented in the next chapter.

Chapter 6

Experimental Evaluation

6.1 Experimental setup

After implementing the system infrastructure, two complementary evaluation campaigns were carried out. The first campaign was executed in a **virtual-machine (VM) environment** and was used as the primary benchmarking platform because it is reproducible, easy to reset, and fast to iterate on during development. The second campaign was executed on **automotive embedded hardware** in order to validate the same framework under realistic deployment conditions.

Keeping these two campaigns conceptually distinct is important for the methodology of this chapter. The **VM campaign** was used for the full automated benchmark and for the generation of the aggregate plots discussed in this chapter. The **automotive-hardware campaign** was used to validate the same orchestration workflow on real target nodes and to compare the order of magnitude of the most relevant metrics with those previously observed in the VM.

6.1.1 Virtual machine environment

The virtual environment was implemented using **Oracle VM VirtualBox** in order to provide an isolated and reproducible testing platform. The guest system runs **Ubuntu Server 22.04 LTS**. The virtual machine was configured with *3 GB of RAM*, *two virtual CPU cores mapped from the host AMD Ryzen 5 3500U processor*, and *16 MB of allocated video memory*.

This virtualized environment is distinct from the one used in Chapter 4 for the orchestrator comparison. The two experimental campaigns were conducted with different VM configurations because they served different evaluation goals: comparative assessment in Chapter 4 and framework-level benchmarking in Chapter 6.

This configuration provides sufficient computational resources to execute the containerized workloads while maintaining a controlled and repeatable environment for automated measurements. In addition, the selected VM hardware specifications are broadly comparable to those typically available in modern automotive embedded

platforms, making the VM suitable as a first-order benchmark environment before deploying the same software stack on the target hardware.

The orchestration platform used in the experiments is *Ankaio* version 0.7.0, running with *Podman* version 4.9.3 as container runtime. Ankaio was installed using the official installation script (`install.sh`) provided by the developers. Using the same installation procedure recommended by the project ensures that the environment can be easily reproduced and allows the same setup to be deployed consistently on the real evaluation platform.

6.1.2 Automotive hardware environment

To evaluate the system in a realistic automotive scenario, an additional validation campaign was conducted on two automotive development boards. These boards are designed for vehicle network processing and service-oriented gateway applications and therefore provide a representative embedded platform for modern automotive computing systems.

Each board is based on the NXP S32G processor family:

- Quad-core **ARM Cortex-A53**
- **Cortex-M7** microcontroller cores
- Dedicated hardware **accelerators for networking**
- **4 GB** of **RAM**
- Multiple automotive communication interfaces including **CAN FD, LIN and FlexRay**
- Several **Gigabit Ethernet** interfaces for high-bandwidth communication

Both boards run a Linux distribution based on the Yocto Project, which is widely used in embedded and automotive environments to build customised operating systems. In order to support the execution of containerized workloads, several Linux kernel components had to be enabled to allow the installation and correct operation of the Podman container runtime. The same software stack used in the VM was reproduced on the boards: Podman version 4.9.3 was used as the container runtime, and Ankaio version 0.7.0 was installed using the official installation script (`install.sh`) provided by the developers.

The two boards were interconnected through an Ethernet switch, enabling communication via IP networking. This configuration allowed the deployment of the distributed orchestration architecture, with the Ankaio server running on one node and the agents running on the connected boards. Access to the boards was performed remotely via SSH through the same Ethernet network. The Atlas project source code was transferred to the devices as a compressed archive and deployed directly on the target systems. This setup allowed the evaluation of the system in a distributed embedded environment closely resembling the networking conditions of an automotive platform.

6.2 Evaluation metrics

The experimental evaluation focuses on assessing the behaviour and performance of the Atlas framework across the two deployment environments described above. The evaluation is structured around five complementary dimensions: **operation latency**, **startup time**, **memory usage**, **reliability**, and **functional validation**.

6.2.1 Operation latency

For each orchestration primitive exposed by Atlas—namely **START**, **STOP**, **MOVE**, and **UPDATE**—the system records the **wall-clock duration** of the corresponding SDK call using Python’s `time.time()` function. The measurement spans the interval from the moment Atlas issues the request through the Ankaaios Python SDK to the instant the SDK returns control to the caller. Every operation is logged to a persistent file (`ankaaios_operations.log`) together with a timestamp, the target workload name, the measured duration, and a success/failure flag. An example log entry is shown below:

```
1 [2025-03-07 14:22:01] START | Workload: nginx | Duration: 0.150s |
   Status: SUCCESS
```

The following statistical indicators are computed from the collected measurements:

- **Average duration** (arithmetic mean)
- **Minimum and maximum duration**
- **Total cumulative duration**
- **Count** of operations executed

These statistics are aggregated at three levels of granularity: (i) *per operation type*, (ii) *per workload*, and (iii) *per workload–operation pair*, enabling a fine-grained analysis of how different workloads behave under each orchestration action.

6.2.2 Startup time

While the operation latency metric captures the round-trip time of the SDK API call, it does not reflect the actual time required for a workload to become fully operational. To address this, the framework implements a dedicated **startup time** measurement that tracks the complete interval from the initial `apply_workload()` invocation until the workload reaches the **RUNNING** execution state as reported by the Ankaaios control plane. The startup time is therefore **cumulative**: it includes the API round-trip time recorded by the **START** metric, followed by the container scheduling, image pull (if the image is not cached locally), container creation, and process initialisation phases.



Figure 6.1: VM benchmark: mean duration per workload and operation

This measurement is performed by the `wait_for_running()` utility function, which polls the Ankaio state at regular intervals (default: every 500 ms) until one of the following conditions is met:

- The workload transitions to the **RUNNING** state, indicating successful startup;
- The workload enters a terminal failure state (e.g. **FAILED**, **REMOVED**);
- A configurable timeout (default: 60 s) expires without a conclusive state transition.

The result is logged as a **STARTUP** entry in the same operations log:

```
1 [2025-03-07 14:22:03] STARTUP | Workload: nginx | Duration: 2.350 s |
  Status: SUCCESS
```

The comparison between the **START** (API round-trip) and **STARTUP** (time-to-RUNNING) durations for the same workload provides insight into the overhead introduced by the container runtime and image management, as opposed to the pure orchestration control-plane latency.

6.2.3 Memory usage

In addition to time-based metrics, the framework captures the **runtime memory consumption** of each deployed workload. After a workload reaches the **RUNNING** state, a short stabilisation period (2 s) is observed, after which the memory usage is sampled by querying the **Podman REST API** through its Unix domain socket.

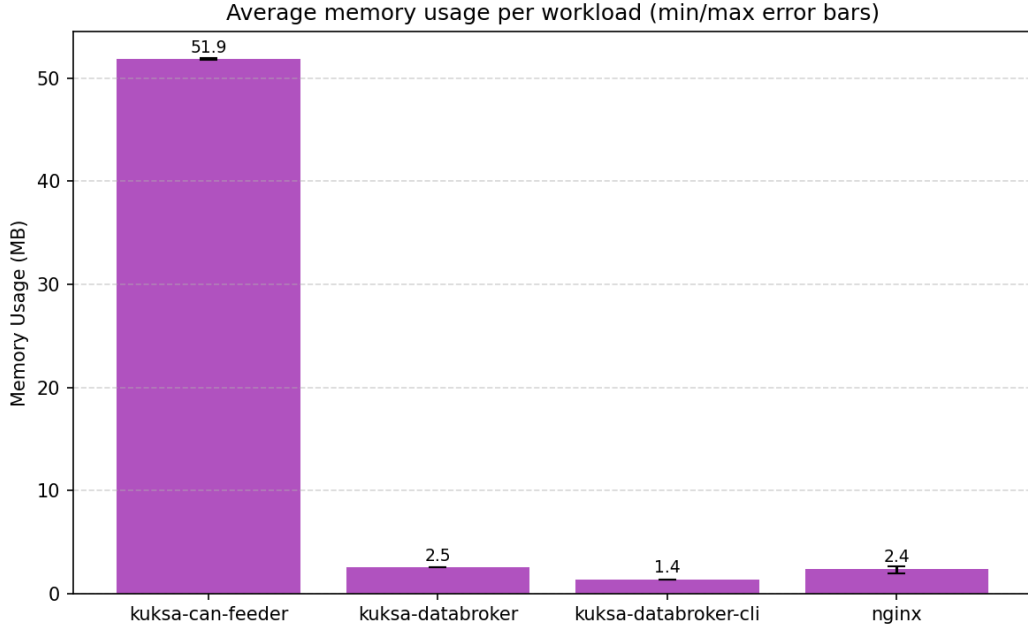


Figure 6.2: VM benchmark: memory usage per workload

The `get_container_memory()` utility function communicates with the Podman `libpod` API endpoint (`/v4.0.0/libpod/containers/{id}/stats`) to retrieve the current resident memory usage of the container, expressed in megabytes. The function implements two lookup strategies: a direct query by workload name, and a fallback approach that first resolves the container identifier via name filter and then retrieves its statistics. This dual strategy accommodates variations in how Podman assigns container names depending on the orchestration context.

To enable this measurement, the Podman Unix socket (`/run/podman/podman.sock`) is bind-mounted into the Atlas container via the volume declaration in the Ankaio startup state file (`state.yaml`). Memory measurements are logged as `MEMORY` entries in a dedicated format within the same log file:

```
1 [14:22:05] MEMORY | Workload: nginx | MemUsageMB: 2.4 | Status: SUCCESS
```

The collected **memory** data is **analysed per workload**, computing the average, minimum, and maximum observed values across all benchmark rounds. In the material retained for this thesis, quantitative memory measurements are available for the VM campaign. On the S32G boards, the Podman installation used for the hardware campaign did not expose the additional bind-mount configuration required by the extended benchmarking instrumentation—most notably the Podman socket mount used by Atlas to query container statistics. For this reason, the retained hardware log does not contain `MEMORY` entries, and memory usage is discussed quantitatively only for the VM campaign.

6.2.4 Reliability and functional validation

Each logged operation carries an outcome label (**SUCCESS**, **FAILED**, or, in the case of update recovery, **ROLLBACK**). The success-to-failure ratio is computed both globally and per operation type, providing an overall measure of system reliability under repeated execution. For **STARTUP**, a failure indicates that the workload did not reach the **RUNNING** state within the configured timeout or entered a terminal failure state before doing so.

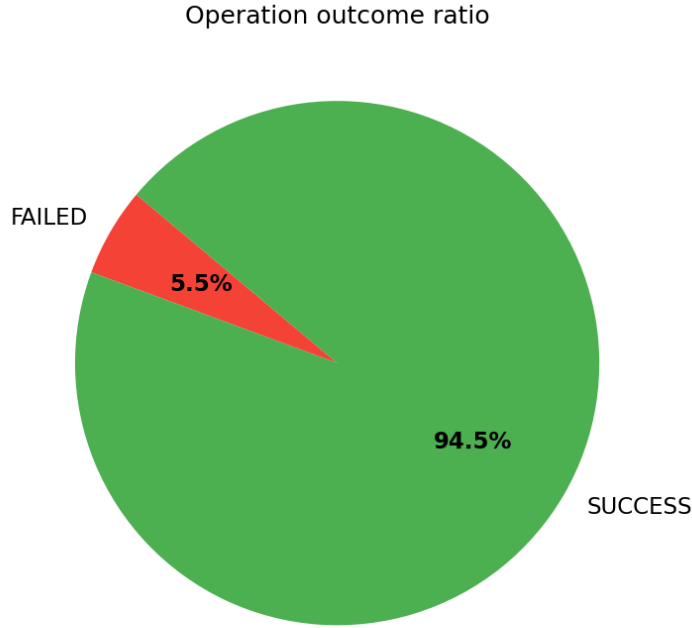


Figure 6.3: VM benchmark: overall outcome distribution

It should be noted that the observed reliability depends not only on the ability of Ankaio to correctly schedule and activate the workloads, but also on the **correctness of the workload definition files** and the availability of the referenced container images. Errors in the state files—such as incorrect image references, malformed command options, or missing volume mounts—will result in failed operations that are attributable to configuration or deployment issues rather than to limitations of the orchestration platform itself. Therefore, the reliability metric should be interpreted as a measure of the **end-to-end system behaviour**, encompassing the orchestrator, the runtime configuration, and the validity of the container images used during the experiment.

In addition to these outcome labels, the evaluation also considers **functional validation**: whether the KUKSA-related workloads can be started, moved, and updated in a distributed deployment, and whether Atlas correctly handles dependency-sensitive recovery actions such as rollback and configuration propagation.

6.2.5 Benchmarking methodology

As described in Section 5.4.1, the benchmarking script automates full orchestration cycles across all workloads and agents. Building upon the base sequence of operations detailed therein, the extended benchmark configuration used for the experimental evaluation also collects **startup time** and **memory usage** measurements at each deployment step. Specifically, after every **START** operation, the benchmark waits for the workload to reach the **RUNNING** state (logging a **STARTUP** entry) and then samples the container memory consumption via the Podman API (logging a **MEMORY** entry). This augmented cycle therefore produces three log entries per deployment—**START**, **STARTUP**, and **MEMORY**—in addition to the **MOVE**, **UPDATE**, and **STOP** entries already described.

The methodology differs slightly between the two evaluation environments:

- **VM campaign.** The full automated benchmark was executed with the default configuration of 5 rounds and a 2s inter-operation pause, resulting in a dataset of several hundred log entries. The VM dataset is the basis of the plots shown in Figures 6.1, 6.2, and 6.3.
- **Automotive-hardware campaign.** On the S32G boards, the same Atlas workflow was exercised on real hardware and the resulting log was retained for quantitative comparison of the most relevant metrics. The retained board log contains repeated **START**, **STARTUP**, **MOVE**, **UPDATE**, and **ROLLBACK** events, which are sufficient to quantify control-plane latency, startup behaviour, relocation latency, and recovery after failed updates. In this hardware setup, however, the Podman installation did not expose the additional bind mounts used by the extended VM benchmark—in particular the Podman socket mount required for runtime inspection from inside Atlas. As a consequence, the retained board dataset does not contain **MEMORY** entries; moreover, the retained subset does not include the full benchmark teardown phase and therefore no **STOP** entries are available for quantitative analysis.

6.2.6 Scope and limitations

While the evaluation framework provides broad coverage of the key performance dimensions, some limitations should be noted.

The **STARTUP** measurement relies on polling the Ankaio state at discrete intervals (500ms by default); consequently, the reported time-to-**RUNNING** has an inherent quantisation error bounded by the polling interval. A finer polling frequency would reduce this error at the cost of increased load on the control plane.

The memory measurement captures a **single-point snapshot** taken shortly after the workload reaches the **RUNNING** state. It does not track memory consumption over the full workload lifetime and therefore may not capture peak usage during transient phases. Continuous memory profiling would require a dedicated monitoring agent, which was outside the scope of this evaluation.

The retained S32G log used in this chapter contains a representative subset of the full hardware campaign rather than the complete automated benchmark dataset available for the VM. In addition, the board setup did not include the extra Podman bind-mount configuration used by the VM benchmark for runtime memory inspection. For this reason, the automotive-hardware analysis in Section 6.3.2 focuses on the metrics directly supported by the retained log: control-plane latency, startup behaviour, migration latency, update recovery, and reliability.

Finally, the framework does not currently measure **CPU usage** or **network I/O**. These metrics, while relevant for a complete performance characterisation, would require additional integration with the container runtime’s statistics API and are left as a direction for future work.

6.3 Results and discussion

The experimental results show that the proposed architecture, based on Eclipse Ankaio, Podman, and the Atlas management framework, can be exercised both in a reproducible VM environment and on real automotive hardware. The key methodological outcome of this chapter is not that the two environments are identical, but that they exhibit **comparable orders of magnitude for the relevant orchestration operations**. This makes the VM a useful benchmark proxy for rapid development and regression testing, while the S32G boards remain necessary for final validation on the target platform.

6.3.1 Results from the VM benchmark campaign

The VM campaign was used for the full automated benchmark and generated the plots already shown in Figures 6.1, 6.2, and 6.3. The main results can be summarised as follows:

- The API round-trip times for the core orchestration primitives remained in the order of **single-digit to low double-digit milliseconds**. In particular, the previously reported values were approximately **7–10 ms** for **START**, **5–15 ms** for **STOP**, and **15–27 ms** for **MOVE**.
- The **UPDATE** operation exhibited higher durations, typically around **1.0–1.4 s**, largely because the current Atlas implementation performs a delete-and-redeploy sequence with an intentional one-second pause between removal of the old instance and deployment of the new one.
- When the required images were already cached locally, container startup times were typically around **1 s**. Higher startup times were observed when the runtime had to pull images from a remote registry.
- As shown in Figure 6.2, the majority of workloads stayed far below the tens-of-megabytes range, while **kuksa-can-feeder** maintained a steady memory footprint around that level.

These VM results provide the baseline against which the automotive-hardware measurements can be interpreted.

6.3.2 Results from the S32G board campaign

The retained log collected on the S32G boards contains **41** operation entries in total. Among them, **36** completed with status **SUCCESS**, **4** with status **FAILED**, and **1** **UPDATE** triggered an explicit **ROLLBACK** sequence. The board dataset includes **START**, **STARTUP**, **MOVE**, **UPDATE**, and **ROLLBACK** entries.

Table 6.1 summarises the main quantitative results extracted from the retained S32G log.

Metric	Samples	Mean	Min	Max
START (control-plane latency)	18	12.33 ms	10 ms	29 ms
MOVE (control-plane latency)	3	19.67 ms	17 ms	21 ms
UPDATE (successful cases only)	2	11.19 s	1.034 s	21.344 s
ROLLBACK (single recovery case)	1	63.600 s	63.600 s	63.600 s
STARTUP (all cases)	16	8.653 s	0.025 s	60.375 s
STARTUP (successful cases only)	12	6.500 s	1.041 s	31.989 s
STARTUP (successful warm starts, < 2 s)	9	1.496 s	1.041 s	1.569 s

Table 6.1: Quantitative results from the retained S32G board log

Several observations can be made from these values.

First, the **START** and **MOVE** operations on the S32G boards remain in the same order of magnitude as those observed on the VM. The mean **START** latency on the boards is **12.33 ms**, while the mean **MOVE** latency is **19.67 ms**. These values are fully compatible with the conclusion that the orchestration control plane itself adds only a small overhead, even on embedded hardware.

Second, the **STARTUP** metric shows a clear distinction between **warm** and **cold** starts. When the image is already available locally and the workload starts successfully without additional recovery actions, the startup time observed on the boards is typically around **1.5 s** (mean of the warm successful subset: **1.496 s**). This is slightly higher than, but still comparable to, the approximately **1 s** startup behaviour discussed for the VM. By contrast, cold starts and problematic launches produce much higher values, up to **31.989 s** for a successful startup and **60.375 s** for a timeout case.

Third, the **UPDATE** metric must be interpreted carefully. In Atlas, an update is not a pure control-plane operation: it includes configuration rewrite, deletion of the current workload instance, a fixed one-second pause, redeployment, and verification of the new instance. Consequently, the successful board updates range from **1.034 s** to **21.344 s**. The best-case value is directly comparable with the VM result, whereas the larger value reflects the cost of redeployment and image availability on the real target.

6.3.3 Cross-environment comparison

Table 6.2 compares the typical values previously observed on the VM with the quantitative results extracted from the S32G boards.

Metric	VM campaign	S32G board campaign
START latency	Typically 7–10 ms	Mean 12.33 ms; range 10–29 ms
STOP latency	Typically 5–15 ms	Not available in the retained board dataset; the hardware setup did not preserve the full teardown phase of the extended benchmark
MOVE latency	Typically 15–27 ms	Mean 19.67 ms; range 17–21 ms
UPDATE duration	Typically 1.0–1.4 s in the benchmark campaign	Best case 1.034 s; successful cases up to 21.344 s; rollback case 63.600 s
STARTUP time	Around 1 s when images are cached; much higher in cold-start cases	Warm successful starts mean 1.496 s; cold/failing cases up to 60.375 s
MEMORY usage	Quantitatively available and charted	Not available on the retained board setup because the Podman socket bind mount required by Atlas for memory sampling was not present

Table 6.2: Comparison between VM benchmark results and S32G board results

The most relevant point in Table 6.2 is that the **control-plane metrics are comparable across environments**. The board measurements are not numerically identical to the VM results, but they remain within the same operational regime: START and MOVE are still completed in a few tens of milliseconds, and successful warm startups remain around one to one and a half seconds. The larger discrepancies appear mainly in operations whose total duration is strongly influenced by external factors such as image pulls, registry access, or failure recovery.

6.3.4 VSS data exchange, controlled relocation, and update recovery

The board campaign also provided functional validation of the distributed KUKSA-based workload set. The retained log shows repeated successful activations of `kuksa-databroker`, `kuksa-databroker-cli`, and `kuksa-can-feeder`, which confirms that the same workload set used in the VM can also be deployed on the S32G platform.

A particularly relevant result concerns workload migration. The retained board log contains three successful MOVE operations for `kuksa-can-feeder`, all completed in **17–21 ms**. This confirms that Atlas can request relocation of a distributed workload between nodes with a small control-plane latency on the target hardware.

The hardware campaign also validates the update-recovery path. One **UPDATE** operation intentionally referenced an invalid image tag and triggered a rollback. Atlas restored the previous image and recorded a dedicated **ROLLBACK** event. This behaviour is important because it demonstrates that the framework does not merely detect a failed update, but can also perform a controlled recovery to the last known working configuration.

Two clarifications are important here. First, the migration result should not be interpreted as a completely transparent relocation at the application level: when the databroker changes location, dependent workloads still require the configuration propagation described in Chapter 5. Second, the update mechanism does **not** provide zero-downtime replacement. In the current prototype, updates are implemented through a controlled delete-and-redeploy sequence and therefore introduce a brief service interruption. The experimental result demonstrates the feasibility of controlled image replacement and recovery, not seamless in-place updates.

6.3.5 Why the VM is still a useful benchmark platform

The S32G measurements strengthen, rather than weaken, the methodological role of the VM. The board campaign shows that the VM does not produce a misleadingly optimistic picture of the control-plane behaviour. For the most relevant orchestration actions, the order of magnitude observed on the VM is preserved on the real hardware:

- **START** remains in the low tens of milliseconds;
- **MOVE** remains in the low tens of milliseconds;
- successful warm **STARTUP** events remain close to one to one and a half seconds.

For this reason, the VM can be considered a **good benchmark proxy for rapid experimentation**. New management features can be developed and regression-tested quickly in the VM, where the environment is easy to reproduce and reset. Once the feature behaves correctly in that environment, it can then be validated on the S32G boards to confirm that the behaviour remains acceptable under realistic embedded constraints.

At the same time, the board results also show why final validation on real hardware remains necessary. The cold-start penalties, image-pull delays, and failure-recovery times observed on the boards are operationally significant and would not be fully characterised by control-plane latency alone. In other words, the VM is a useful development and benchmarking environment, but it is **not a substitute** for target-hardware validation.

6.3.6 Summary

The experimental evaluation supports four main conclusions.

- The Atlas–Ankaaios stack introduces only a **small control-plane overhead**. On the VM, **START**, **STOP**, and **MOVE** remained within the range already discussed by the benchmark results; on the S32G boards, **START** and **MOVE** remained respectively at **12.33 ms** and **19.67 ms** on average.
- **Successful warm startups are comparable across environments**. The VM showed startup times around **1 s** when images were cached, while the S32G boards showed a mean of **1.496 s** for successful warm starts.
- **Update latency is dominated by redeployment and recovery behaviour rather than by pure orchestration overhead**. The best successful update on the boards completed in **1.034 s**, while a failed update followed by rollback required **63.600 s** because the framework waited for the failing instance to time out before reverting.
- The **VM is a credible benchmark environment for rapid testing of new features**, because the relevant control-plane metrics remain comparable when the same framework is deployed on the S32G boards. However, final validation on the target hardware remains necessary for cold-start, networking, and failure-recovery behaviour.

These results provide a stronger methodological basis for the conclusions of the thesis: the VM is useful for fast and repeatable benchmarking during development, while the automotive boards are essential for validating the same design under realistic deployment constraints.

Chapter 7

Conclusions

This thesis set out to investigate whether a lightweight, container-based software architecture could satisfy the requirements of modern automotive embedded systems—specifically enabling flexible application deployment while maintaining low orchestration overhead. The following sections summarise the objectives that have been achieved, the limitations that remain, and a selection of practical extensions based on the proof-of-concept implementation.

7.1 Achieved Objectives

The primary outcome of this work is a *functional proof of concept* (PoC) demonstrating end-to-end workload orchestration on automotive-grade hardware. The system—comprising the Atlas workload manager, Eclipse Ankaio as the container orchestrator, and Podman as the OCI-compliant runtime—was deployed on two NXP S32G development boards equipped with quad-core ARM Cortex-A53 processors and 4 GB of RAM.

The key objectives achieved are:

1. **Hardware–software decoupling.** By packaging each application component as a container image, the deployment of workloads becomes less dependent on the physical ECU topology. At the same time, container images must still be built for the target architecture (ARM) and therefore cannot be directly reused from x86 builds.
2. **Dynamic lifecycle management.** Atlas exposes a command-line interface to start, stop, move, update, and monitor workloads at runtime. The experimental results demonstrated controlled workload relocation and redeployment across nodes through updates to the workload specification. This confirms the feasibility of flexible workload allocation across multiple ECUs, although the mechanism does not constitute live migration, since it introduces a transient downtime and may require reconfiguration of dependent applications.
3. **VSS-based data exchange.** The integration of the KUKSA databroker, CAN feeder, and databroker CLI demonstrates that the Vehicle Signal Specification

can serve as a standard data layer, enabling signal routing across workloads regardless of their physical placement.

4. **Local container image update and rollback.** The `update` command modifies a workload’s container image and performs a controlled delete-and-redeploy procedure with brief downtime. This demonstrates a locally orchestrated software update mechanism with rollback support, rather than a complete OTA update pipeline.
5. **Low orchestration overhead.** Benchmarking shows that lifecycle API operations execute with limited latency, typically within single-digit to low double-digit milliseconds. By contrast, workload startup times were substantially higher, indicating that the main delays are likely related to image retrieval and workload redeployment rather than to the orchestrator control plane itself. In the VM campaign, memory usage of individual workloads remained modest. Although equivalent memory measurements were not available on the S32G boards because of instrumentation limitations, the VM results still provide useful supporting evidence, as the latency trends observed in the VM and on the real hardware were comparable for the other benchmarked metrics.

Taken together, these results indicate that Ankaaios, combined with Podman and a thin management layer such as Atlas, provides a viable foundation for workload orchestration on resource-constrained automotive platforms.

7.2 Limitations

Several limitations remain in the current implementation:

Service discovery. Workloads communicate through manually configured IP addresses or `localhost`, without an automated discovery mechanism. When a workload is migrated, its consumers must be reconfigured manually. A lightweight service registry or DNS layer would be required in multi-node deployments.

Dependency management. Workload dependencies are declared statically at registration time with only three coarse-grained conditions (`ADD_COND_RUNNING`, `ADD_COND_SUCCEEDED`, `ADD_COND_FAILED`). Circular dependencies are not detected, and runtime re-evaluation of the dependency graph is not supported.

Volatile session state. Workloads dynamically loaded at runtime are stored only in memory (`AVAILABLE_WORKLOADS` dictionary). Restarting Atlas results in loss of these definitions. A persistent registry would be needed for long-running deployments.

Resource limits. No CPU or memory constraints are enforced on containers. In a production automotive system, unbounded resource consumption could interfere with critical functions.

7.3 Possible Extensions

Based on the PoC, the following practical extensions could be pursued with moderate effort:

- **Persistent workload registry.** Replace the in-memory dictionary with a file-based or key-value store to retain dynamically loaded workloads across container restarts.
- **Continuous resource monitoring.** Implement periodic CPU, memory, and network monitoring to support trend analysis and capacity planning.
- **Workload offloading.** Enable migration of non-critical workloads to external edge nodes while preserving data flows via the VSS layer.

In summary, this thesis has demonstrated the feasibility of a lightweight software stack for orchestrating containerized workloads in a constrained automotive setting. The combination of Podman as container runtime, Ankaaios as orchestrator, and the Atlas management layer built on top of the available SDKs provides a practical basis for managing distributed workloads in a zonal architecture while reducing coupling between applications and hardware-specific signal interfaces. The proof of concept and the associated benchmarking tools do not constitute a production-ready platform, but they establish a reproducible foundation for further investigation of software-defined vehicle architectures.

Appendix A

Supporting Code Listings and Analysis Environment Setup

This appendix reports two representative code excerpts from Atlas and the Vagrantfile used to build the analysis environment for the comparative evaluation of the orchestration frameworks.

```
1 Vagrant.configure("2") do |config|
2   config.vm.box="ubuntu/jammy64"
3   orchestrators = {
4     "bluechi" => 2201,
5     "ankaaios" => 2202,
6     "k3s" => 2203
7   }
8   orchestrators.each do |name, port|
9     config.vm.define name do |node|
10      node.vm.hostname = "#{name}-vm"
11      node.vm.network "forwarded_port", guest: 22, host: port, id: "ssh"
12
13      node.vm.provider "virtualbox" do |vb|
14        vb.name = "#{name}-vm"
15        vb.cpus = 2
16        vb.memory = 2048
17      end
18      node.vm.provision "ansible_local" do |ansible|
19        ansible.playbook = "ansible/playbook.yml"
20        ansible.extra_vars = { orchestrator: name }
21      end
22    end
23  end
```

Listing 3: Vagrantfile snippet illustrating the setup

```
1 workload_info = AVAILABLE_WORKLOADS[name_workload]
2 config_file = workload_info['config']
3 dependencies = workload_info['dependencies']
4
5 # Build the workload specification
6 builder = WorkloadBuilder() \
7     .workload_name(name_workload) \
8     .agent_name(target_agent) \
9     .runtime("podman") \
10    .restart_policy(restart_policy) \
11    .runtime_config_from_file(config_file)
12
13 # Inject declared dependencies
14 for dep in dependencies:
15     builder = builder.add_dependency(
16         dep['name'], dep['condition']
17     )
18
19 # Submit to the Ankaaios control plane
20 workload = builder.build()
21 ankaaios.apply_workload(workload)
```

Listing 4: Workload creation through the Ankaaios SDK

```
1 # Resolve the YAML path relative to the shared volume
2 if os.path.isabs(yaml_arg):
3     yaml_path = yaml_arg
4 else:
5     yaml_path = os.path.join(WORKLOADS_DIR, yaml_arg)
6
7 # Register the new workload in the in-memory catalogue
8 AVAILABLE_WORKLOADS[workload_name] = {
9     'config': yaml_path,
10    'dependencies': dependencies,
11 }
12
13 # Build the workload specification from the YAML file
14 builder = WorkloadBuilder() \
15     .workload_name(workload_name) \
16     .agent_name(target_agent) \
17     .runtime('podman') \
18     .restart_policy('NEVER') \
19     .runtime_config_from_file(yaml_path)
20
21 # Inject any declared inter-workload dependencies
22 for dep in dependencies:
23     builder = builder.add_dependency(
24         dep['name'], dep['condition']
25     )
26
27 # Submit to the Ankaaios control plane
28 workload = builder.build()
29 ankaaios.apply_workload(workload)
```

Listing 5: Dynamic workload registration through the Ankaaios SDK

Bibliography

- [1] CNCF (Cloud Native Computing Foundation). *K3s: Lightweight Kubernetes*. The certified Kubernetes distribution built for IoT and Edge computing. 2024. URL: <https://k3s.io/> (visited on 05/22/2024) (cit. on p. 1).
- [2] Eclipse Foundation. *Eclipse Bluechi Documentation*. A systemd service controller for multi-node environments. 2024. URL: <https://bluechi.readthedocs.io/en/latest/> (visited on 05/22/2024) (cit. on p. 1).
- [3] Eclipse Foundation. *Eclipse Ankaaios Documentation*. Workload orchestrator for automotive software. Eclipse Foundation. 2024. URL: <https://eclipse-ankaaios.github.io/ankaaios/latest/> (visited on 05/22/2024) (cit. on p. 1).
- [4] COVESA Project Contributors. *Vehicle Signal Specification (VSS)*. Accessed: 2026-03-15. 2026. URL: <https://covesa.global/project/vehicle-signal-specification> (cit. on p. 1).
- [5] Y. Wang, H. Zhang, X. Li, et al. “Review of Electrical and Electronic Architectures for Autonomous Vehicles: Topologies, Networking and Simulators”. In: *Automotive Innovation* (2024). DOI: 10.1007/s42154-023-00266-9. URL: <https://doi.org/10.1007/s42154-023-00266-9> (cit. on p. 3).
- [6] Arun T. Vemuri. *Processing the Advantages of Zone Architecture in Automotive*. <https://www.ti.com/document-viewer/lit/html/SSZT211>. Texas Instruments, Technical Article. 2023 (cit. on p. 4).
- [7] ETAuto Editorial Team. *Automakers shift to zonal architecture for efficiencies*. Accessed: 2026-03-15. 2024. URL: <https://auto.economictimes.indiatimes.com/news/auto-technology/automakers-shift-to-zonal-architecture-for-efficiencies/118844882> (cit. on pp. 5, 6).
- [8] Stephanie Susnjara and Ian Smalley. *What Is Container Orchestration?* Accessed: 2026-03-01. IBM. 2024. URL: <https://www.ibm.com/think/topics/container-orchestration> (cit. on p. 10).
- [9] Eclipse Foundation. *Eclipse Kuksa*. Accessed: 2026-03-15. 2026. URL: <https://projects.eclipse.org/projects/automotive.kuksa> (cit. on p. 15).
- [10] Rakuten Cloud. *Introduction to cloud-native orchestrator: From A (automation) to Z (zero touch)*. Accessed: 2026-03-02. Dec. 2025. URL: <https://cloud.rakuten.com/blogs/introduction-to-cloud-native-orchestrator-from-a-automation-to-z-zero-touch> (cit. on p. 20).

- [11] K3s Project Authors. *K3s Documentation: Architecture*. Accessed: 2026-03-15. 2026. URL: <https://docs.k3s.io/architecture> (cit. on p. 27).
- [12] Eclipse BlueChi Project Contributors. *Eclipse BlueChi: Architecture*. Accessed: 2026-03-15. 2026. URL: <https://bluechi.readthedocs.io/en/latest/architecture> (cit. on p. 31).
- [13] Eclipse Ankaaios Project Contributors. *Eclipse Ankaaios: Architecture*. Accessed: 2026-03-15. 2026. URL: <https://eclipse-ankaaios.github.io/ankaaios/latest/architecture> (cit. on p. 33).
- [14] Ankaaios Project. *Ankaaios Python SDK*. <https://github.com/eclipse-ankaaios/ank-sdk-python>. Accessed: 2026-02-18 (cit. on p. 44).
- [15] Podman Project. *Podman Documentation*. <https://docs.podman.io/>. Accessed: 2026-02-19.
- [16] Eclipse Foundation. *Eclipse Software Defined Vehicle*. <https://projects.eclipse.org/working-group/eclipse-software-defined-vehicle>. Accessed: 2026-02-18.
- [17] Nand Kochhar. *Understanding Software-Defined Vehicles (SDVs)*. Siemens Automotive & Transportation. Oct. 2025. URL: <https://blogs.sw.siemens.com/automotive-transportation/2025/10/08/understanding-software-defined-vehicles/>.