Can Software Containerisation Fit The Car On-Board Systems ?

David Fernández Blanco, Frédéric Le Mouël, Trista Lin, Amir Rekik

Can Software Containerisation Fit The Car On-Board Systems ?

David Fernández Blanco*†, Frédéric Le Mouël*, Trista Lin† and Amir Rekik†
*Univ Lyon, INSA LYON, Inria, CITI, EA3720, 69621,Villeurbanne, France.
†STELLANTIS, 78140, Velizy-Villacoublay, France.
Corresponding author: david.fernandez-blanco@insa-lyon.fr

***Abstract*—As the automotive industry evolves towards interconnected and intelligent vehicles, the integration of complex electronic and software components has become paramount. However, this increased complexity brings new challenges in ensuring system safety, security, and life-cycle management. In this article, we focus on virtualisation (virt.), particularly containerisation (cont.), as a solution to mitigate integration stress in multi-node environments. We present a detailed review of the automotive constraints and ecosystem, along with the selection criteria for virt. technologies, justifying the scope of our study. Our main contribution is a two-phased evaluation of cont. tools. Firstly, we assess popular single-node container engines (Docker, Containerd, and Linux Containers LXC) based on CPU, RAM, and file I/O overhead over multiple hardware configurations. Secondly, we evaluate their multi-node scalability. The results show that Docker performs in average better than the other solutions for automotive on-board architectures.**

***Index Terms*—Containerisation, Automotive ICT systems, Micro-processor Units (MPUs), Multi-node, Dynamicity**

I. Introduction

In recent decades, Information and Communication Technologies (ICT) have played a dominant role in transforming the automotive industry and seamlessly integrating it into Smart City ecosystems. This transformation has significantly enhanced vehicle connectivity and computational power, leading to the emergence of new services such as AI-based Advanced Driver Assistance Systems (ADAS) or customizable In-Vehicle Infotainment (IVI) systems. Furthermore, recent technological advancements such as Over-The-Air (OTA) software updates, partial vehicle operation, or software-defined vehicle variants have stimulated automakers to explore more flexible and dynamic software business opportunities.

However, in their pursuit of innovation, automakers have primarily focused on introducing new features to enhance passenger experience, continuously adding more resources to meet the growing computing demands. Unfortunately, this approach has not been accompanied by corresponding evolutions in software architecture and control panel design. As a result, system complexity has significantly increased [1], giving rise to numerous challenges [2] related to system safety, security, flexibility, and software life-cycle management. Consequently, software development, integration, and maintenance costs have escalated, hindering the automakers' objectives. Additionally, with the increasing adoption of fleet-control Vehicle-to-Everything (V2X) applications, in-vehicle systems need to allocate additional resources to coordinate with their surroundings, particularly for ADAS purposes. However, as shown in §III, these applications may not require continuous operation. Therefore, the ability to dynamically instantiate the necessary software applications presents an intriguing opportunity for resource optimisation and cost reduction.

Traditionally, to mitigate integration stress and achieve full isolation of software contexts, the IT domain recommends implementing a common middleware. This middleware handles installation, integration, and run-time maintenance procedures, consolidating the complexity into the development of this shared framework instead of each individual application. By adopting this approach, although it may entail higher initial costs, the risk of conflicts is minimized, and the system's safety, dynamism, and security are ensured in a flexible and simplified manner [3]. These frameworks typically rely on virt., a widely used technique. However, there are two main types of virt.: Hardware Abstraction Layer (HAL) level virt. and Operating System (OS) level virt.. While both can be suitable for automotive in-vehicle systems [4], their contexts of use differ significantly. HAL level virt. excels in providing high levels of security and isolation by creating fully isolated software environments with their own operating systems, drivers, and allocated hardware resources. On the other hand, OS level virt. offers greater resource efficiency, flexibility, scalability, and dynamism, albeit with some trade-offs in security and isolation. In this paper, we will primarily focus on OS level virt. as it aligns better with the objectives of flexibility and dynamism for automakers.

Within the context of this paper, we will propose a performance analysis of the most widely used OS level tools, from now on called containerisation frameworks, such as Docker [5], [6], Kubernetes [7], and Linux Containers (LXC and LXD) [8]. We will assess their individual performance when operating in a single-node and their collective performance when operating within a cluster of nodes. Additionally, it's worth noting that for this study, we will be using automotive-like hardware and orchestrating it following near future Zonal Architecture patterns. Thus, this paper proposes:

– A detailed description of the current automotive context, architecture, and challenges hindering the adoption of OS-level virt. within automotive in-vehicle systems (cf. § II-A). Followed by a comprehensive survey of virt. techniques, specifically focusing on the different OS-level virt. engines (cf. § II-B).

– A realistic use-case scenario highlighting the significance of software dynamism and flexibility in in-vehicle systems, justifying the emphasis on OS-level virt. over HAL-level virt. and showing relevant the performance metrics, supported by scientific literature and innovation reports from automakers (cf. § III).
– A detailed performance analysis (cf. § V) of the above-mentioned cont. solutions, in single node and cluster scenarios over varied automotive-like hardware.
– A discussion on the suitability of containerisation for the automotive context based on the obtained results, along with an identification of remaining challenges (cf. § VI).

This paper is organised as follows: Section II explores the automotive and virt. context. Section III presents a detailed use-case scenario illustrating the importance of software dynamism and flexibility in future vehicles, along with the performance metrics to study. Section IV describes the testing procedures and experiment choices. Section V presents and analyses the experiment results. Section VI discusses the performance analysis results, their limitations, and proposes future works for virt. in automotive systems. Finally, Section VII synthesises the paper findings.

II. Background

This section provides an overview of the automotive and virt. backgrounds, exploring key developments and their intersection in shaping the future of mobility. By understanding the evolution of these domains, we can uncover the potential at the intersection of cars and virt. technologies.

A. Automotive Background

In this subsection, we explore the causes behind the current automotive IT evolution. We then delve into two main aspects of in-vehicle architectures: the automotive software environment and the Electrical/Electronic (E/E) architecture.

1) Evolution Causes: Societal changes have triggered a significant transformation in the automotive industry. Younger generations are more inclined towards mobility options rather than owning cars, relying heavily on shared and on-demand services [9]. Simultaneously, there has been a decline in interest in driving as it is now perceived as a mundane task, given the advancements in autonomous driving systems and V2X applications [10], [11]. Consequently, the traditional notion of vehicles as symbols of status has shifted towards a focus on utility and software customisation. Accordingly, the automotive software architecture must adapt to effectively manage the increasingly diverse passenger preferences and the dynamic needs of V2X applications.

2) Automotive Software Environment: Within the automotive ecosystem, there exists a diverse and collaborative environment where automakers play the role of integrating various software components developed by different suppliers. Thus, automakers are in charge of managing the centralised software hub, update campaigns, security certificates, etc. On the one hand, the evolution of software development methodologies in the automotive industry has been driven by increasing complexity and the need to meet quality, safety, and efficiency requirements. It has transitioned from traditional waterfall to more agile approaches like Automotive SPICE [12]. However, as this standard predates the software boom in automotive systems, it lacks modern software dynamism and has high hardware-software coupling, complicating innovation. New initiatives seek to merge IT-oriented Agile models with automotive requirements. On the other hand, the evolution of business use-cases and the enhancement of in-vehicle computing and connectivity has led to the creation of new application profiles [13]. These include collaborative V2X services, highly dynamic and computed in multiple vehicles and infrastructures, enhanced IVI services with high networking and computing capabilities for specific scenarios, and remote services with direct cloud interaction, impacting integration pipelines and software life-cycle management.

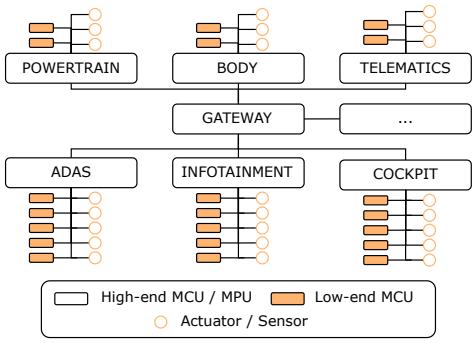


Fig. 1. Current Automotive Domain Vehicle E/E Architecture.

3) In-vehicle E/E architectures: As mentioned earlier, the increasing software functionalities in cars have led to a rise in in-vehicle computing power, resulting in a significant increase in embedded ECUs. This has led to unmanageable system complexity and challenges in keeping up with app. demands for computing, network, and energy consumption [2], [14]. Consequently, the E/E architecture has evolved, transitioning from a fully distributed architecture composed mostly of mono-functional micro-controllers (MCUs) connected through low-efficient networks like Controller Area Network (CAN), Local Interconnect Network (LIN), or FlexRay. Modern architectures, inspired by the Domain Vehicle E/E reference architecture (cf. Fig. 1), employ fewer, more powerful microprocessors (MPUs) primarily connected via Ethernet. This trend has reduced the price, weight, space, and complexity of these systems. In near future, systems are likely to evolve to be more centralised as in Central Computer Arch. or Zonal Arch., where one or a few higher-end MPUs would encompass all functions within a sub-region or even the entire vehicle.

B. Virtualisation Background

Virt. has been traditionally employed to tackle issues like inflexibility, lack of dynamism, software integration, and management challenges in modern automotive systems. In this subsection, we'll examine various types of virt., their pros and

cons. Then, we'll focus on the most suitable technique for the automotive context: OS Level virt., as shown in Table I.

1) Virtualisation levels: Virt. is a technology that combines or divides computing resources to create one or multiple operating environments. It enables hardware and software partitioning, aggregation, simulation, emulation, and time-sharing. This technique allows a single Electronic Control Unit (ECU) to run sets of code independently and securely in isolation. Virt. enhances hardware/software decoupling, software modularity, and provides the necessary flexibility, security, and dynamism required by automotive systems (cf. § III). It also facilitates environment maintenance, reduces the cost of software sand-boxing, and enables testing and simulating complex real-world scenarios, all of which are highly desired by automakers. However, virt. can be implemented at various levels with varying performance characteristics, which are:

- Instruction Set Architecture (ISA) level virt.: ISA serves as an interface between hardware and software, defining CPU control, data types, registers, and memory management. At this level, virt. interprets instructions one by one, translating source ISA instructions into host machine-compatible ones. This allows running legacy code on newer hardware but introduces significant system overhead. Some examples are Bochs and Crusoe.
- Hardware Abstraction Layer (HAL) level virt.: In an effort to reduce the interpretation latency associated with ISA level virt., HAL level virt. aims to leverage the architectural similarities between systems. It maps virtual resources to physical ones and uses native hardware for computations within the virtual machine (VM) through a hypervisor middleware. It can be implemented making all the guest OSs independent from each other (i.e., full-virt.) either over the physical hardware (bare-metal) or over an existing OS (hosted) or modifying the guest OS kernel to act as a bridge between applications and hardware resources to improve performance slightly (i.e. para-virt.). Some examples are Linux KVM or XEN for full-virt. and Lguest for para-virt.
- Operating System (OS) level virt.: OS level virt. creates an abstraction layer between the OS and user apps, enabling isolated app containers with server-like functionality, improved efficiency, and scalability. Cont., the fundamental unit, operate on a Linux-based host OS with a cont. control engine. Unlike hypervisor-based solutions, OS level virt. doesn't require individual guest OSs per each app. Instead, the host kernel runs multiple isolated user-space instances with necessary libraries and binaries added. This approach offers higher performance, faster boot times, and reduced resource overhead. OS level virt. incorporates isolation mechanisms, resource management, network abstraction, and more. Some examples are Docker, LXC/LXD, Rkt, and Podman.
- Library level virt.: Library level virt. relies on APIs provided by user-level library implementations, abstracting OS-specific details for ease of use. It creates a virtual environment above the OS layer, exposing different but equivalent binary interfaces to emulate required application binary interfaces and program APIs. Some examples are WINE, WABI, and LxRun.
- Application level virt.: App. level virt. runs an application as if it were a virtual machine. The virt. middleware operates as an app. process on top of the OS, providing an abstraction of a VM that executes programs written in a specific abstract machine definition language. Some examples are Java VM and Parrot.

When comparing the automotive evolution causes and trends discussed in § II-A to the different virt. layers in Table I, OS-level virt. stands out as offering superior efficiency, flexibility, and scalability. Consequently, it is the most suitable choice for achieving the automotive objectives in software dynamism and flexibility. However, OS-level virt. raises concerns regarding software security and isolation properties.

2) Container run-times: Having highlighted the benefits of cont. in the previous subsection, we now delve into the core of this technology: the container engines (or run-times). These engines are responsible for executing and managing containers over the host OS, interacting with the OS kernel to dynamically isolate and allocate resources. The performance of the cont. solution critically depends on this software block. In this section, we will explore the main cont. engines, upon which we will base our study later on.

- *Linux Containers (LXC) [8].* LXC was the pioneering comprehensive containerisation solution. It creates and manages multiple isolated Linux Virtual Environments on a standard Linux kernel using cgroups and Linux namespaces, avoiding hardware preloading emulation or ad-

TABLE I
COMPARISON OF DIFFERENT VIRTUALISATION LAYERS.

Layer	Type	Safety Possibilities	Isolation	Efficiency	Complexity	Flexibility	Scaling	Boot Time
ISA Level		Low	Medium	Medium	Medium	High	Medium	Medium
HAL Level	Bare-metal hypervisor	High	High	High	Medium	Medium	Medium	Low
	Hosted hypervisor	Medium	Medium	Medium	Medium	Medium	Medium	Medium
	Para-virtualisation	Medium	Medium	High	High	Medium	High	Low
OS Level		Low	Low	Very High	Medium	Very High	Very High	Very Low
Library Level		Low	Low	Low	Low	Low	Low	High
Application Level		-	Low	Medium	-	-	-	Medium

ditional overhead. To enhance LXC and incorporate features from other container solutions, LXD was introduced as a container manager built on top of LXC, providing an improved UX with a control daemon accessible through a REST API based on liblxc.

– *Containerd [5], [6].* Containerd, the default run-time used by Docker, follows the Open Container Initiative (OCI) standard specification through runc implementation. It aims to provide minimal application virt. capabilities while facilitating image and snapshot storage, along with execution contexts. Containerd dynamically allocates host resources among containers based on their application requirements. Containerd's versatility extends beyond Docker, as it is also employed as a container runtime in other solutions like Kubernetes.
– *CRI-O [7].* CRI-O is a standardised interface designed for Kubernetes plugins responsible for managing and monitoring containers. It establishes a stable communication interface between kubelet (the primary node agent of Kubernetes) and the host container run-time. CRI-O uses gRPC, a cross-language library for remote procedure calls using Protocol Buffers, to connect OCI run-times with higher-level Kubernetes components. The interaction lifecycle with CRI-O is similar to containerd, utilising the CRI endpoint, and runc serves as the default OCI runtime when using CRI-O.
– *CoreOS RKT [15].* RKT is a cont. run-time developed by CoreOS, offering enhanced security through strong signature verification for image downloads and invocations. It also reduces the authorisation level required to manage containers, improving system security. RKT implements mechanisms for auditing containers, making it valuable for critical systems like those in automobiles. Its composability allows compatibility with traditional service management tools and higher-level orchestrators like Kubernetes. However, as the RKT project has been recently abandoned and lacks ARM architecture support, it is unsuitable for the automotive systems.

C. Automotive containerisation open issues

Even though cont. has gained significant popularity and adoption in software development and deployment, its implementation within cars and other industrial sectors is still relatively new and faces several ongoing concerns.

Resource Constraints. Cars have limited computing resources compared to data centers or cloud environments. Cont. involves running multiple apps. within isolated environments, which can consume additional CPU, memory, and storage. Optimising resource allocation and managing container density while meeting performance requirements is a significant challenge [16] in maintaining system efficiency while offering dynamicity and flexibility.

Hardware and Platform Diversity. The automotive industry includes various car manufacturers, models, and embedded systems with diverse hardware architectures and platforms. Ensuring cont. compatibility and adaptability across this ecosystem (legacy, present, future...) presents challenges in standardisation, portability, and cross-platform support [17].

Safety and Security. These properties are critical in the automotive industry, especially for in-vehicle systems. While cont. offer advantages in software integration and system dynamism, it also faces significant security issues [18]. Addressing them is essential to prevent potential exploits and vulnerabilities without adding complexity.

Certification and Regulatory Compliance. Automobiles undergo strict certification procedures to meet safety and regulatory standards such as ISO-26262 or ISO-21434. The adoption of cont. in car systems may require the development of new certification frameworks and standards tailored to address the unique challenges and risks posed by cont. environments [19].

III. Use Case Scenario

The goal of this section is to show the importance of software flexibility and dynamicity, and how these attributes are closely tied to the system's performance and resource utilisation. The scenario takes place within the context of a smart city ecosystem, where an autonomous vehicle navigates through different situations, leveraging various applications and services. This scenario is part of a wider study realised in collaboration with STELLANTIS[1]. Thus, we will focus on the first two situations of that scenario, as they are sufficient to match our purposes, which are:

– *Leaving the parking lot.* When leaving the parking lot, the vehicle gains access to the network, allowing seamless integration with real-time traffic data and communication with nearby cars and pedestrians. The efficiency of the system's CPU, memory utilisation, and file I/O operations are essential for real-time data processing and communication, enabling the identification of potential threats and accident prevention. Furthermore, the ability to dynamically allocate resources and handle mutexes efficiently becomes crucial as multiple apps, such as navigation and music streaming, are initiated simultaneously.
– *Going through an intersection.* When approaching an intersection, the vehicle needs to communicate with the driving infrastructure, specifically the traffic light, to obtain additional information for safe navigation, initiating now new inactive apps. This communication relies on V2X apps and requires the system to handle locks efficiently, ensuring smooth coordination between different components. Besides, as the vehicle has internet access, it is able to download and build new apps. or updates.

Throughout the use-case scenario, the system's performance and efficiency are assessed using selected test metrics that reflect the demands of in-vehicle archs. These metrics include CPU, memory access, file I/O operations, multi-threading, mutex handling, image build efficiency and container start time. By evaluating these metrics, we can gauge the system's ability to meet the performance requirements.

[1]The detailed use-case scenario and benchmark are accessible at: https://github.com/DavidFdezBlanco/containerisation_performance_analysis.git

IV. Materials and Methods

In light of the evolving trends in E/E architecture, which suggest a notable decline in highly-constrained MCUs within vehicle archs. in favour of more advanced MPUs, this study specifically concentrates on the latter. This section provides an overview of the experiment objectives, hardware/software setup and benchmark employed for our perf. study.

A. Experiment objectives

Performance profiling and evaluation have gained increasing importance in the container research community. Earlier studies primarily compared containerisation pioneers like Docker and LXC with traditional HAL virtualisation solutions such as XEN or KVM on server-like devices [20]–[23]. More recent research expanded the comparison to newer containerisation tools, including RKT [15], LXD [24], and Kubernetes [25]. However, there remains a gap in performance evaluations focusing on multi-node extensions of these container solutions. Only a few examples, such as [26] comparing Kubernetes variants on IoT nodes and [27] comparing Docker Swarm to Kubernetes on cloud-like servers, exist. As a result, our paper aims to address three unexplored research gaps in the field:

- The suitability study of widely used containerisation engines (Docker, LXC/LXD, and Kubernetes) for automotive-like nodes in singleton architectures.
- The investigation of the impact of automotive-like hardware variants (legacy nodes and future improvements) on container engine performance in singleton architectures.
- The examination of the impact of multi-node clustering on container engine performance, this time using Docker Swarm, Kubernetes, and LXD Clustering extensions.

Further details on the automotive-like hardware selection and software configuration are given next subsection.

B. Experimental Setup

1) Hardware Configuration: The tests were conducted on a publicly-accessible Fog/Edge platform [28]. For the automotive-like nodes, we utilised the Raspberry Pi 3B, which features an ARM Cortex-A53 chip-set similar to those deployed in current vehicle systems [2]. Legacy nodes are composed of Raspberry Pi 2 boards, bearing an ARM Cortex-A7 chip-set, representing a previous automotive chip-set generation. For future nodes, we selected the latest Raspberry Pi 4, as we expect future chip-sets to follow in the footsteps of Raspberry generations. A comprehensive description of these three boards is available in Table II. All these cards had a 32GB Micro-SD card SDCS2 Class 10 storage and are interconnected through Ethernet backbone (Cat. 6 cables) forming a mesh as in Zonal Architectures.

TABLE II
Hardware detailed description

	CPU			RAM	
Card model	Chipset	Cores	Clock-speed	Size	Generation
Raspberry Pi 2	ARM Cortex-A7 (32-bit)	4	0.9 GHz	1 GB	LPDDR2
Raspberry Pi 3B	ARM Cortex-A53 (64-bit)	4	1.2 GHz	1 GB	LPDDR2
Raspberry Pi 4	ARM8 Cortex-A72 (64-bit)	4	1.5 GHz	4 GB	LPDDR4

2) Software Configuration: Table III provides an overview of the software configurations employed in the experiment, including the host operating systems and container engine versions. However, when utilising Kubernetes (k3s) on Raspberry Pi 2 and Raspberry Pi 3B with the Ubuntu Server 22.04 host OS, constant memory swapping significantly impacted the functionality of k3s. Therefore, a lighter OS (RPI OS Lite v6.1 Bullseye 64-bits) was used for conducting these tests.

TABLE III
Software detailed description

Host OS	Ubuntu Server 22.04 lts 64-bits
Docker	v24.0.2
Kubernetes (k3s)	v1.27.3
LXC / LXD	v5.0.2

C. Benchmark Specification

Regarding the evaluation criteria, in conjunction with the use-case scenario in §III, all concurs on the significance of assessing CPU performance, memory utilisation, and File I/O operations. Some also emphasise evaluating mutex synchronisation and multi-threading scheduling performance. Moreover, when comparing to HAL level virtualisation, certain papers focus on system boot time, provisioning, and fail-over time. However, the selection of benchmarks varies among the papers. The most commonly used benchmarks include Sysbench (https://github.com/akopytov/sysbench), Phoronix (https://www.phoronix-test-suite.com/), and OCCT (https://www.ocbase.com/). For our performance experiment, we opted for Sysbench as our comparative benchmark. Although we considered Phoronix and OCCT, they proved either incompatible or exhibited inconsistent behavior on ARM architectures. From Sysbench, we considered several benchmarks that align with most of the metrics described in the use case scenario. These benchmarks include the CPU stress benchmark, the memory access benchmark, the filesystem-level benchmark, the thread-based scheduler benchmark, and the POSIX mutex benchmark. However, since Sysbench does not include some important properties specifically relevant to the automotive context, we chose to extend Sysbench with custom container-oriented tests. These additional tests were developed to measure the image build and start time, allowing us to assess the efficiency of the application installation and launch processes. We also included a test for error detection reactivity, which measures the responsiveness of the containerisation system in identifying and handling errors or faults within the containerised environment. It's important to note that to ensure experiment repeatability, the complete benchmark used and the results are accessible in our repository.

V. Results

The section presents the findings of our performance evaluation and analysis of cont. technologies for automotive on-board

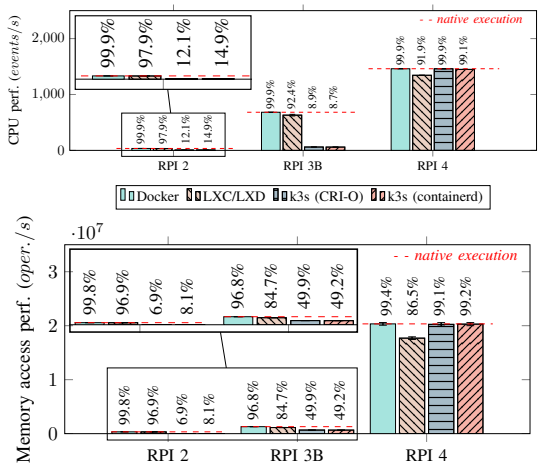
systems. The results are divided into two main sections. The first section focuses on the suitability of cont. for automotive on-board systems in singleton archs., covering the two primary objectives outlined in §IV-A. The second section explores the impact of clustering on container engine performance, which matches the third aforementioned objective.

A. Container Suitability in Singleton Architectures

For this experiment, we used three distinct hardware configurations, each not interconnected with the others. These nodes were chosen to represent the past, present, and future of automotive nodes, as described in §IV-B1.

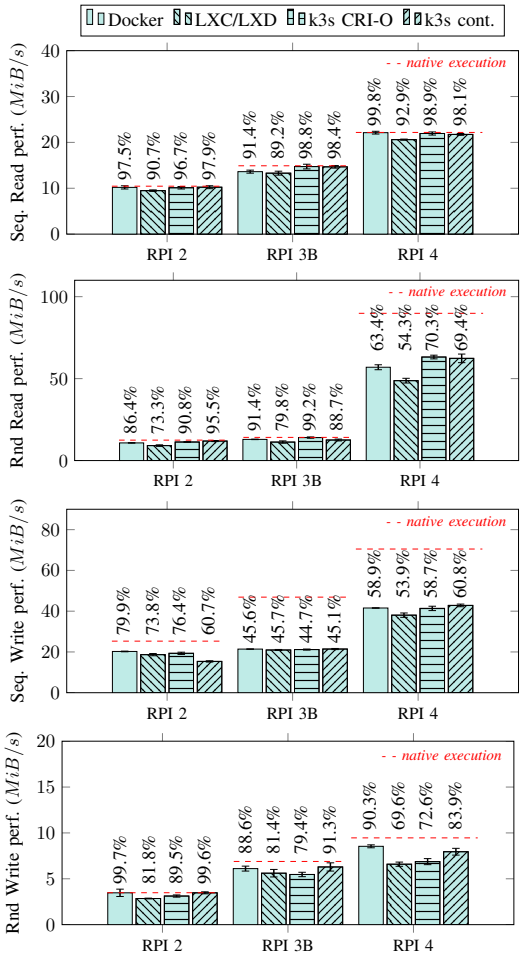
1) CPU Stress & Memory Access Benchmarks: With regard to CPU stress benchmark (Fig. 2 - top), Docker performs exceptionally well with an overhead of less than 0.1% across all hardware configurations. LXC/LXD follows closely as the second-best performer, with a low overhead of less than 8.1% for all configurations. However, Kubernetes shows poor performance on current and legacy hardware, with approximately 90% overhead and significant memory swap, even with Raspbian. Interestingly, Kubernetes performs significantly better on future hardware, approaching Docker's performance and slightly outperforming it when using the CRI-O configuration. With regard to Memory access performance benchmark (Fig. 2 - bottom), Docker remains the top performer. The ranking between LXC/LXD and Kubernetes configurations is less clear. LXC/LXD performs similarly to Docker on legacy hardware but shows slightly higher overhead (approximately 15%) compared to the CPU stress benchmark on actual and future hardware. Kubernetes exhibits reduced overhead (approximately 50%) on actual hardware, and its performance on future hardware approaches that of Docker.

Fig. 2. CPU and Memory access performance over heterogeneous hardware.



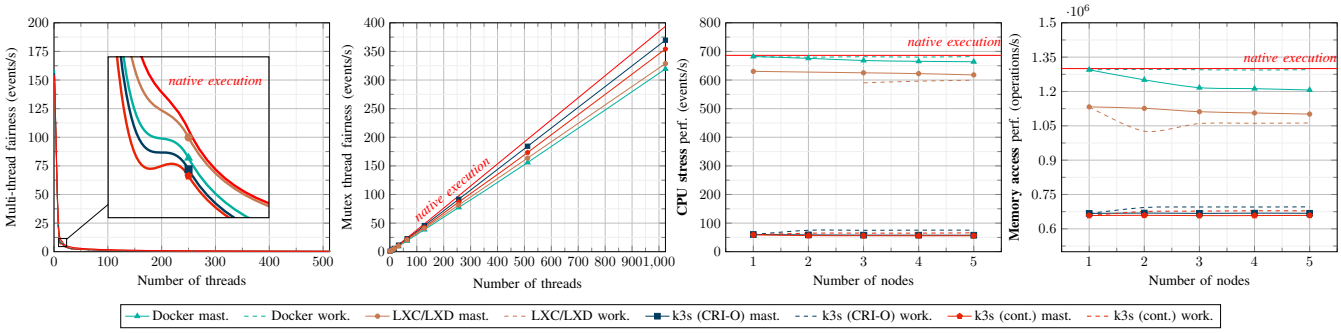
2) File I/O Benchmarks: Fig. 4 presents the results for the sequential and random read and write benchmarks. These figures demonstrate that all the solutions achieve similar performance, with Kubernetes, particularly when using the CRI-O engine, slightly outperforming the other three solutions. Interestingly, for the first and only time, we observe a higher overhead in future hardware configurations compared to current hardware. However, despite the higher overhead, the mean speed is still higher for the future hardware configuration, indicating that overall perf. is better on the newer hardware.

Fig. 3. File I/O performance over heterogeneous hardware.



3) Multi-threading Scheduler & POSIX Mutex Management Benchmarks: These two tests evaluate different aspects but share a reliance on interacting with system locks. In the multi-threading scheduler bench. (Fig. 4 - left), each worker thread is assigned a mutex. During execution, it repeatedly takes the lock, yields it (requesting the scheduler to suspend its execution and place it back at the end of the run-queue), and then unlocks it when scheduled again. This process assesses the multi-threading scheduler's performance and its handling of concurrent locking mechanisms. LXC performs the best in this test, followed by Docker and then Kubernetes. On the other hand, for the POSIX mutex workload (Fig. 4 - medium left), the system application runs a single request per thread. Each request involves taking a random lock, incrementing a global variable, and then releasing the lock. This benchmark focuses on evaluating the performance of the system's handling of POSIX mutexes and its ability to manage concurrent access to shared resources. This test is not influenced by the memory swap issues of kubernetes, being the two configurations of this last the most performant solutions, followed by LXC/LXD and Docker. The results for both legacy and future hardware

Fig. 4. Multi-threading and Mutex Mgmt. in singleton (two left) and CPU Stress and Memory Access in multi-node (two right) both over current hardware.



for these two tests (not presented given the page limit) were similar to those with current hardware.

TABLE IV
BUILD, UPDATE & START BENCHMARK RESULTS.

	Container Engine	Build time (s)	Update time (s)	Start time (s)
Raspberry PI 2	Native execution	$0.3426 \pm 0.83\%$	$0.3411 \pm 0.37\%$	$0.0221 \pm 1.91\%$
	Docker	$112.55 \pm 0.94\%$	$112.23 \pm 1.25\%$ (no cache)	$2.12 \pm 1.61\%$
			$3.36 \pm 1.64\%$ (cached)	
	LXC/LXD	$558.18 \pm 0.98\%$	$554.90 \pm 1.07\%$	$4.42 \pm 1.05\%$
	k3s (CRI-O)	$85.67 \pm 1.92\%$	$74.86 \pm 1.52\%$	$1.95 \pm 0.34\%$
	k3s (containerd)	$96.03 \pm 1.07\%$	$95.47 \pm 1.04\%$	$2.59 \pm 2.18\%$
Raspberry PI 3B	Native execution	$0.3245 \pm 0.59\%$	$0.3211 \pm 0.70\%$	$0.0132 \pm 1.92\%$
	Docker	$79.98 \pm 1.34\%$	$79.83 \pm 1.18\%$ (no cache)	$2.17 \pm 1.24\%$
			$2.97 \pm 1.57\%$ (cached)	
	LXC/LXD	$494.90 \pm 0.62\%$	$493.10 \pm 0.65\%$	$3.9001 \pm 1.36\%$
	k3s (CRI-O)	$60.7765 \pm 1.71\%$	$60.4787 \pm 1.04\%$	$1.5558 \pm 0.61\%$
	k3s (containerd)	$77.5373 \pm 1.48\%$	$77.4710 \pm 1.81\%$	$2.0891 \pm 1.05\%$
Raspberry PI 4	Native execution	$0.26 \pm 0.07\%$	$0.26 \pm 0.07\%$	$0.97 \pm 1.54\%$
	Docker	$50.46 \pm 1.50\%$	$50.20 \pm 0.72\%$ (no cache)	$2.12 \pm 1.61\%$
			$1.31 \pm 1.47\%$ (cached)	
	LXC/LXD	$388.43 \pm 0.27\%$	$379.85 \pm 0.22\%$	$3.12 \pm 1.94\%$
	k3s (CRI-O)	$46.92 \pm 0.98\%$	$43.89 \pm 0.34\%$	$0.99 \pm 0.17\%$
	k3s (containerd)	$51.23 \pm 1.14\%$	$44.83 \pm 1.49\%$	$1.29 \pm 1.28\%$

4) Build and Start time Benchmarks: Software dynamicity is a critical feature for enabling reactivity within automotive systems. The tests were conducted over three apps: a ADAS-like function, a real-time function, and a IVI-like function. The results of the tests (cf. Table IV) showed that the nature of the app had no significant influence on the build, update, and start times. This means that regardless of the complexity or size of the app, the overhead for both build and start times was substantial and not satisfactory for meeting the real-time dynamicity objectives required in automotive systems. Besides, the performance metrics revealed that Kubernetes with CRI-O achieved the best results among the tested options, followed by Docker and Kubernetes with containerd. However, LXC/LXD's performance was notably poorer compared to the other three container technologies. Although the container engine start and build times need improvement, they can still be integrated into on-board systems by initiating applications pre-emptively based on various context information such as maps, trajectory, and other relevant data.

B. Container Suitability in Multi-node Architectures

In this section, after analysing and presenting the results for singleton architectures, the focus shifted to multi-node architectures. After conducting the same tests as in the precedent scenario without significant changes between them, we decided to present only two specific tests: the CPU stress test and the memory access test. The results of these tests over the current hardware configuration are depicted in Fig. 4 - medium right & right. A significant difference was observed between LXC/LXD and the other three solutions (Kubernetes with CRI-O, Docker, and Kubernetes with containerd). In the case of LXC/LXD, the performance of their worker nodes was impacted, whereas for the other three solutions, this impact was directly felt by the master node. Considering automotive architectures, the researchers concluded that centralising the impact on the master node is a better approach, especially since the master node is likely to be the most powerful one in the on-board systems. As for the other three solutions, although there was an impact on the performance of the master node, it was deemed acceptable, with a relatively low overhead of around 10-15% for small architectures like in-vehicle systems. Importantly, the performance of the worker nodes remained unaffected regardless of the number of nodes.

VI. Discussion & Future Works

In this discussion section, we will analyse and interpret the results to understand the implications for the automotive industry and potential areas for improvement.

Effectiveness of Containerisation Technologies Nowadays. Docker excelled in benchmarks with minimal overhead and high performance. LXC/LXD followed closely behind, while Kubernetes showed subpar performance with notable overhead and memory swap. Choosing the right technology is crucial for optimal results, though start-time may still pose challenges for real-time dynamicity requirements in some use cases.

Real-world Performance vs. Future Expectations. The study emphasised hardware's role in cont. performance. Kubernetes showed promising improvements on future hardware, bridging the gap with Docker and even surpassing it with CRI-O. Yet,

for current and legacy hardware, it proved less efficient, urging thoughtful cont. decisions for automotive applications.

Scalability: The multi-node study provides valuable insights for the automotive industry adopting cont. in multi-node architectures. Understanding the impact on master and worker nodes helps make informed decisions on the best cont. technology. Notably, the multi-node impact is mainly observed on master node performance for most cont. options.

Despite these encouraging results, there remain several open issues and research challenges that need to be addressed in the near future to fully incorporate containers into on-board architectures. One crucial area is investigating new ways to enhance real-time invocation and reactivity in cont. environments, which is of utmost importance for ADAS. Additionally, exploring hybrid cont. architectures that combine traditional real-time systems with cont. applications could be a promising avenue. Moreover, developing a virt. framework for MCU nodes, which are more cost-effective than MPU nodes and commonly used in legacy systems, would be highly beneficial during the transition phases. Another area of interest is exploring the potential of seamlessly integrating cont. and synchronisation with edge computing and fog computing architectures, leveraging their capabilities to optimise performance and resource utilisation. Besides, it is essential to bolster container security by adding further layers of protection, such as hardware isolation or trusted computing, to ensure the integrity of the environments they run in. Finally, standardisation and regulations are crucial aspects to consider, especially for the automotive industry. Establishing a company-wide standard for containers can significantly expedite the software innovation ecosystem, fostering collaboration and compatibility.

VII. Conclusion

In this paper, we presented a performance study of containerisation for automotive-like hardware architectures. This study sheds light on the strengths and weaknesses of different containerisation technologies in varying on-board hardware configurations. The findings shows that Docker emerged as a top performer, achieving low overhead for every test-case in any hardware configuration. LXC/LXD followed closely, showcasing its potential dealing with mutli-threading scheduling. While Kubernetes struggled on current and legacy hardware, it showed promise on future hardware, especially with the CRI-O configuration. However, even though in terms of resource-consumption containers look that they can be a good fit for the automotive industry, bringing flexibility, reusability and reducing considerably the integration efforts, the current frameworks are not yet prepared to deal with the real-time invocation dynamicity wish by the automakers.

VIII. Acknowledgement

The research leading to the results presented in this study was supported by Stellantis under the collaborative framework OpenLab VAT@Lyon, involving STELLANTIS and CITI Lab (ANRT contract n°2020/1415).

References

[1] R. Charette, "How Software Is Eating the Car," in *IEEE Spectrum*, 2021.
[2] D. Fernández Blanco, F. Le Mouël, T. Lin, and M. Escudié, "A comprehensive survey on Software as a Service (SaaS) transformation for the automotive systems," in *IEEE Access*, 2023.
[3] A. Rashid and A. Chaturvedi, "Virtualization and its role in cloud computing environment," in *International Journal of Computer Sciences and Engineering*, 2019.
[4] A. Abuabdo and Z. A. Al-Sharif, "Virtualization vs. containerization: Towards a multithreaded performance evaluation approach," in *IEEE/ACS International Conference on Computer Systems and Applications*, 2019.
[5] Docker, "Empowering App Development for Developers," Available online at https://www.docker.com, 2021.
[6] Containerd, "An industry-standard container runtime with an emphasis on simplicity, robustness and portability," Available online at https://containerd.io/, 2022.
[7] I. Donca and C. Corches, "Autoscaled rabbitmq kubernetes cluster on single-board computers," in *IEEE International Conference on Automation, Quality and Testing, Robotics*, 2020.
[8] S. Kumaran, "Practical LXC and LXD: linux containers for virtualization and orchestration." Apress, 2017.
[9] L. Eliot, "The reasons why millennials aren't as car crazed as baby boomers, and how self-driving cars fit in. forbes," 2019.
[10] F. Kröger, "Automated driving in its social, historical and cultural contexts," in *Autonomous Driving*. Springer, 2016.
[11] I. Panagiotopoulos and G. Dimitrakopoulos, "An empirical investigation on consumers' intentions towards autonomous driving," in *Transportation research part C: emerging technologies*. Elsevier, 2018.
[12] M. Noureldin, "Blending agile methodologies to support automotive spice compliance," *Journal of Software: Evolution and Process*, 2022.
[13] M. Cakır and T. Häckel, "A qos aware approach to service-oriented communication in future automotive networks," in *IEEE Vehicular Networking Conference*, 2019.
[14] M. Traub, A. Maier, and K. L. Barbehön, "Future automotive architecture and the impact of it trends," in *IEEE Software*, 2017.
[15] A. Bhardwaj and C. R. Krishna, "Virtualization in cloud computing: Moving from hypervisor to containerization—a survey," in *Arabian Journal for Science and Engineering*. Springer, 2021.
[16] M. Haeberle and F. Heimgaertner, "Softwarization of automotive e/e architectures: A software-defined networking approach," in *IEEE Vehicular Networking Conference*, 2020.
[17] N. Nayak and D. Grewe, "Automotive container orchestration: Requirements, challenges and open directions," in *IEEE Vehicular Networking Conference*, 2023.
[18] O. Bentaleb and A. Belloum, "Containerization technologies: Taxonomies, applications and challenges," in *The Journal of Supercomputing*. Springer, 2022.
[19] H. Yu and C. Lin, "Automotive software certification: current status and challenges," in *Journal of passenger cars-electronic and electrical systems*. SAE International, 2016.
[20] M. Xavier and M. Neves, "Performance evaluation of container-based virtualization for high performance computing environments," in *IEEE Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, 2013.
[21] M. J. Scheepers, "Virtualization and containerization of application infrastructure: A comparison," in *Twente student conference on IT*, 2014.
[22] E. Casalicchio and V. Perciballi, "Measuring docker performance: What a mess!!!" in *ACM/SPEC International Conference on Performance Engineering Companion*, 2017.
[23] B. Rad and H. Bhatti, "An introduction to docker and analysis of its performance," *Journal of Computer Science and Network Security*, 2017.
[24] A. Putri and R. Munadi, "Performance analysis of multi services on container docker, lxc, and lxd," *Electrical Eng. and Informatics*, 2020.
[25] Y. Mao and Y. Fu, "Resource management schemes for cloud-native platforms with computing containers of docker and kubernetes," *arXiv preprint arXiv:2010.10350*, 2020.
[26] S. Böhm and G. Wirtz, "Profiling lightweight container platforms: Microk8s and k3s in comparison to kubernetes." in *ZEUS*, 2021.
[27] I. Al Jawarneh and P. Bellavista, "Container orchestration engines: A thorough functional and performance comparison," in *IEEE International Conference on Communications*, 2019.
[28] F. Le Mouël, "YOUPI: retour d'expérience et travaux futurs sur une plateforme Fog/Edge Computing," in *Journées Cloud - GDR RSD*, 2019.