Vehicle Integration Platforms Enable Hyperintegration

By Maarten Koning, Wind River Fellow

In automobiles, the amount of data generated, stored and collected – and the number of applications deployed within vehicles to process all this content – has increased dramatically over the last decade. Due to digital transformation, the modern automobile is now a supercomputer on wheels. Vehicle workloads can run concurrently on today's multi-service **integration platforms** thanks to high-powered silicon that's vital to mobile computation platforms.

Continuous Integration (CI), Continuous Deployment (CD) and Continuous Testing (CT) are practices that enable more frequent, lower-ceremony release of the software payloads that are activated as services and applications on a vehicle platform. With CI/CD/CT, a lot of the heavy lifting is done up-front during the software development process which enables these payloads to be taken forward in a largely automated fashion using tooling.

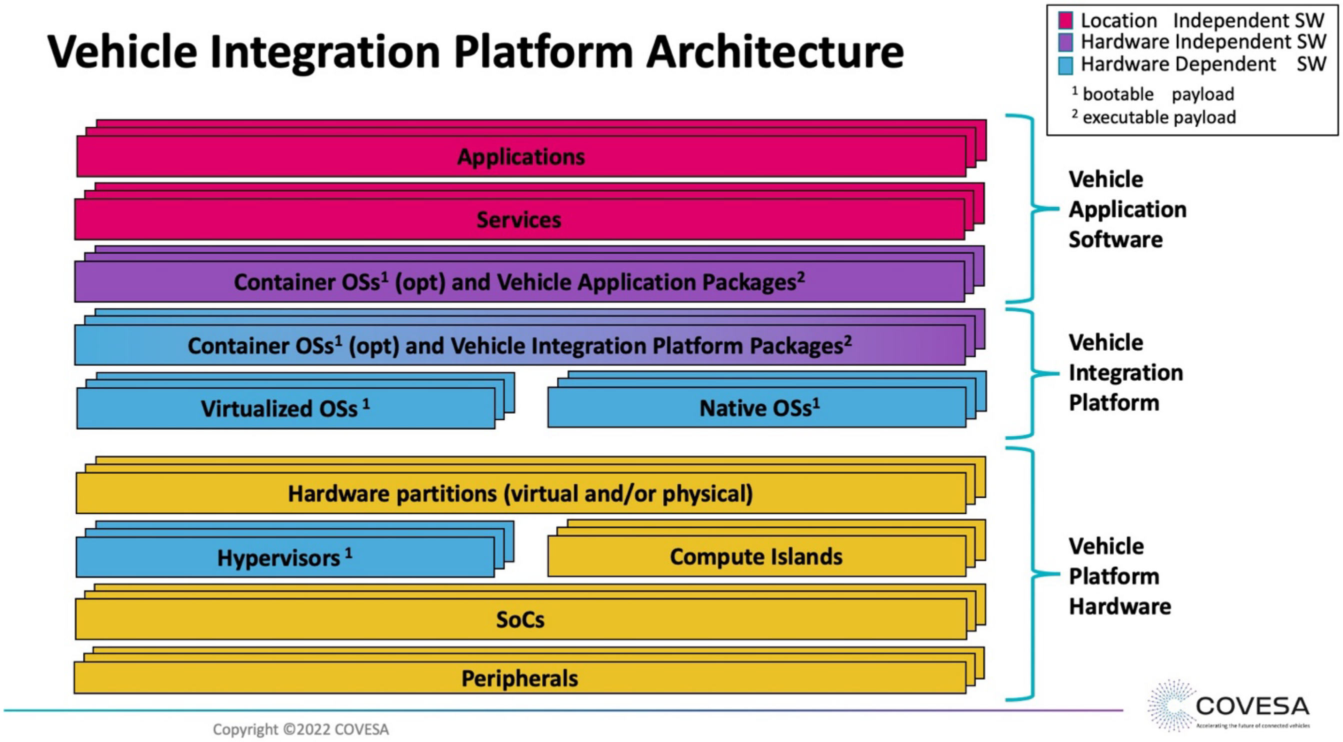
In a complex system like a vehicle, we want the granularity of those released software payloads to be smaller than an entire system to parallelize CI/CD/CT. This is done using **element separation.** Element separation is a software architecture best practice for many reasons as it helps with:

1. Preventing **fault propagation** so we know which payload caused a software (SW) failure.
2. **Provisioning** of compute and memory **resources** so we can engineer the system.
3. **Simplifying implementation** for SW teams by separating functions from each other.
4. **Securing credentials** to the time and place needed - the "least privilege principle."
5. Providing various producers of SW (e.g., ISVs) their own **private execution environment.**
6. Easier management of **software lifecycle** using granular independent SW elements.
7. Containment of software with **specialized core values** (e.g., safety or real-time SW).
8. **Reuse of software** elements across systems, projects and hardware.
9. **Workload orchestration** and optimization such as SW **load balancing** and **scaling.**

These separated software elements are the vehicle applications and services that are activated as one or more multi-threaded processes which comprise the vehicle workloads. It is becoming increasingly helpful to wrap one or more of these services and applications into discrete operating systems (OS) containers so that they can avoid interference from other services and applications with which they don't need to be tightly coupled. One of the advantages of containerizing services and applications is that they can use, and be delivered with, an OS that is composed of the optimal set of files, libraries and support services for their needs.

Host operating systems run on physical or virtual hardware whereas containers share an underlying host OS – although they don't see one other and so they 'think' they have their own OS instance. This is not unlike when multiple host operating systems run on the same CPU cluster using a hypervisor, as in both cases the applications see what looks like their own OS instance and OS object namespace. This practice of containerizing payloads so they each have their own logical OS instance helps reduce interference between payloads and between the underlying host OS and the payloads themselves as well.

One could extend this notion of providing applications and services their own OS instance to compute islands, since they provide a hardware mechanism for OS separation in SoCs without requiring virtualization or container technology to do it. Whether virtualization, containers or compute islands are used to enable services and applications to have their own OS instance, these payloads can integrate, collaborate and be managed similarly with the right platform software. This can be drawn like this:


Vehicle Integration Platform Architecture
🟥 Location Independent SW
🟪 Hardware Independent SW
🟦 Hardware Dependent SW
1 bootable payload
2 executable payload
Applications
Services
Container OSs1 (opt) and Vehicle Application Packages2
Container OSs1 (opt) and Vehicle Integration Platform Packages2
Virtualized OSs 1
Native OSs1
Hardware partitions (virtual and/or physical)
Hypervisors 1
Compute Islands
SoCs
Peripherals
Vehicle Application Software
Vehicle Integration Platform
Vehicle Platform Hardware
COVESA
Accelerating the future of connected vehicles

Since payloads can be combined into working systems from multiple sources, it is helpful to have standardized ways to secure, deliver, deploy and manage them. To do that we have to define the touchpoints between these payloads, standardize those interfaces, and standardize the management actions that can be taken upon those payloads within the vehicle.

This standardization will allow automotive systems to be **assembled** from **ready-made** software payloads within the automotive ecosystem regardless of whether they are in-house, third-party or open source payloads. There are many initiatives within the automotive industry to create such multi-service integration platforms, which I will collectively label Vehicle Integration Platform (VIP) architecture. Many top automotive companies have already announced VIP technologies. Even though some of those VIP initiatives are referred to as an OS, they offer much more than just an operating system. VIPs include capabilities on top as an integration and management infrastructure that helps connect and orchestrate vehicles while also separating vehicle services and vehicle applications from the various execution environments and hardware that the VIP abstracts.

For VIP systems to be able to do their job, they need to be able to process software payloads from various sources including open source, in-house teams and also from third-party ISVs and sub-contractors. One thing that is missing is a way to describe these software payloads in a standard way for a VIP to consume it. For example, if a payload offered self-describing metadata that told the VIP what its resource requirements were (e.g., memory, compute, services, reactivity, etc.) and what deployment models it supports (e.g., migration, hitless software update, suspend/resume, etc.), then the VIP would be able to learn how to integrate and run that payload automatically.

With such metadata, the VIP software would be able to predict if the system can afford to run a vehicle service or vehicle application given the available vehicle resources – without trying and failing, or without denying resources to some other possibly higher-priority application. Once we do this, we will be a step closer to achieving software ***hyperintegration,*** which is highly-automated software integration.

This is the type of challenge that is best solved under the guidance and collective expertise offered through an industry alliance such as COVESA. Together, companies can create a standard that enables low-code/no-code system hyperintegration of software from a standardized automotive ecosystem. This allows companies building new VIP-based vehicles to integrate vehicle extensions, services and applications from other organizations – and the converse is that it allows ISVs to provide specialized ready-made software that can easily be integrated into any VIP-based vehicle.

Taking this road – everyone wins.