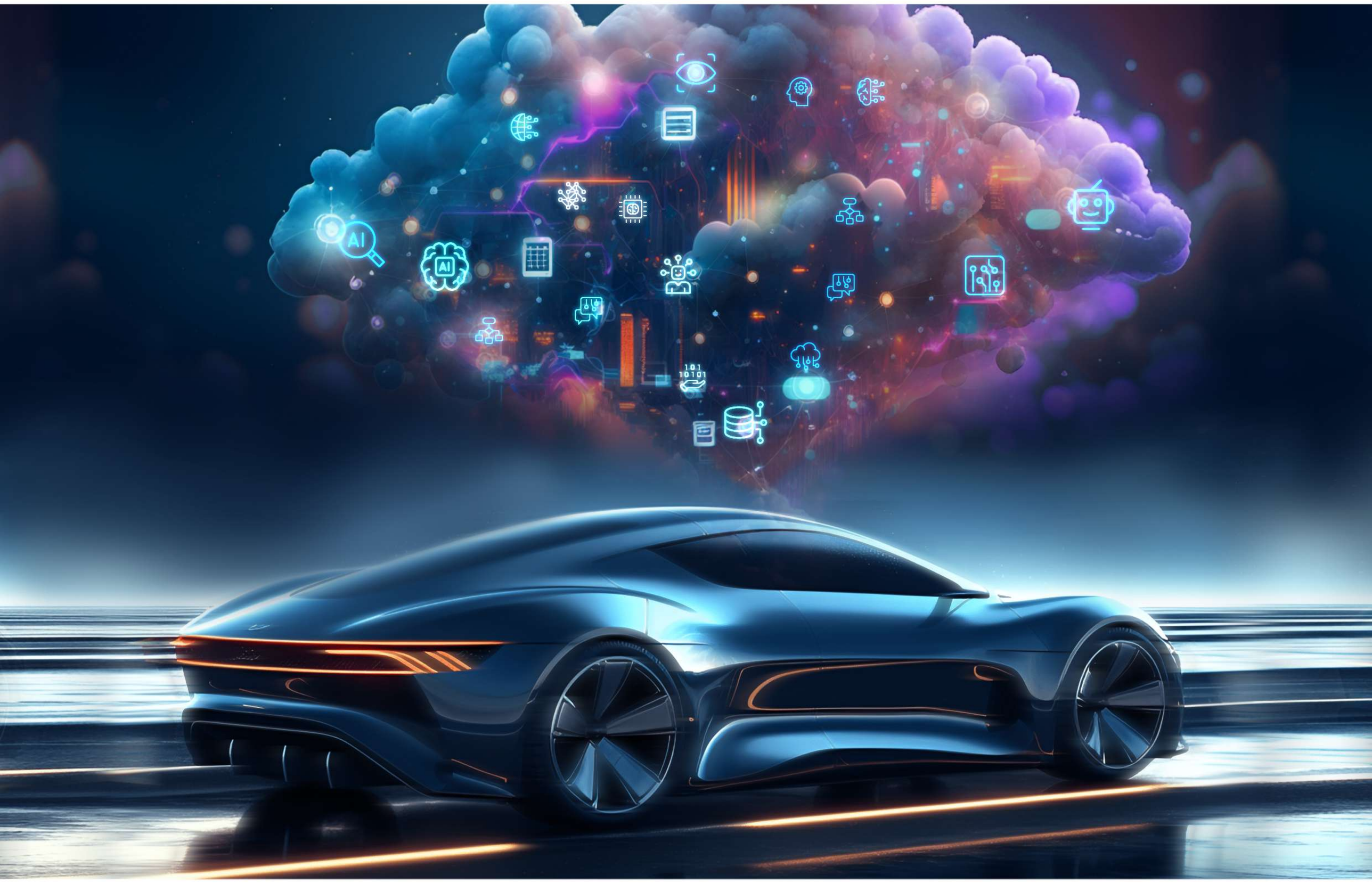
Luxoft
A DXC Technology Company



SDV Actors: In search of a better microservices environment for software-defined vehicles

by **Andre Podnozov**, Chief Architect

Abstract

This is the next installment in a series of articles on SDV.Ops – our approach to developing and operating software-defined vehicles (SDVs). In the first part, we discussed the agility challenge for SDVs, and then we introduced the concept of SDV.Ops for solving that challenge. Now let's explore in more detail one of the fundamental principles of SDV.Ops: A flexible and efficient in-vehicle execution environment. While there are several interesting options for an execution environment, we chose the Actor model of computation as the basis for this exploration.

What do we mean by "Microservices"?

The success of the Agile software development movement that started in the early 2000s was fueled to a large extent by microservices. In its narrow meaning, this word refers to microservices architecture, but with a broader view, the microservice approach to building cloud solutions consists of several aspects:

- **Architecture**: Breaking down large monolithic applications into smaller independent services
- **Process**: Using Agile and DevOps methodologies
- **Technology stack**: Execution environment based on Docker containers and Kubernetes orchestration, in which microservices can deploy and run

In this article, we will focus on the third item above — the execution environment for microservices. Let's take inspiration from cloud-based microservices and try to imagine the technological underpinnings for running in-vehicle microservices.

Benefits of container-based microservices

Containerization brought several significant benefits to microservice implementations:

- **Lower overhead than VMs.** Before containers, solutions were built using Virtual Machines (VMs), which were more resource-hungry, both in terms of storage (VM disk images measured in GBs) and in terms of memory footprint (again, requiring GBs of RAM per VM). Containers are a couple of orders of magnitude more frugal than VMs, allowing many more containers to run on the same computing resources
- **Just enough isolation.** VMs offer the optimal level of isolation needed to protect different cloud customers from each other while their VMs are running on the same underlying hardware of the cloud provider. Containers offer more lightweight isolation, which is appropriate because containers within a solution belong to the same owner so there is no malicious threat from one container to another. As a trade-off for this lower level of isolation, you get the lower container overhead as discussed above.
- **Heterogeneous toolsets.** Container packaging allows agile teams to mix and match different development toolsets (such as programming languages or frameworks) within the same solution.

Given the popularity of container-in-the-cloud solutions, it was only natural to try to adopt them for the in-vehicle environment. There would be a host of obvious benefits, including:

- The wide availability of container-in-the-cloud expertise promises easy sourcing of talent for SDV development, as opposed to the more specialized skillset required for traditional automotive software frameworks

- Rich container toolsets could be seamlessly extended from the cloud into the vehicle for continuous integration and deployment of vehicle features and updates
- Container orchestration in the vehicle could improve the reliability characteristics of vehicle functionality

Downsides of using containers for in-vehicle microservices

While in-vehicle containerization does deliver on many of its promised benefits, it still has significant shortcomings that make it insufficient as an execution environment for in-vehicle microservices. This is not surprising though, because current container technologies were developed for a different set of objectives and constraints, while targeting a different environment — the cloud. Containers in the cloud can enjoy limitless availability of compute resources, with constant connectivity that is very fast and virtually free — unlike the in-vehicle environment. What's more, container orchestration in the cloud was designed for executing compute tasks under much less stringent time and quality constraints than in-vehicle processing.

The main drawback that makes containers a less-than-ideal choice of in-vehicle microservice environment is the level of isolation. The same level of isolation that was considered lightweight on the cloud compared to VMs is now too strong for the in-vehicle microservices purpose. And just like in the cloud, you pay for this unneeded isolation with extra overhead, which manifests in several ways:

- **Higher memory consumption.** A running Docker container can easily add tens of megabytes of memory footprint to even the smallest of microservices. Such an overhead may be negligible in a cloud solution because you can always add a few extra megabytes of memory for literally a penny per month, but the in-vehicle environment has a fixed amount of memory and there is no easy way to add more
- **Architectural degradation.** In order to minimize the total negative impact of the per-microservice overhead, you will have to design the system with a small number of large microservices. This is obviously a wrong path to take as the microservices would be regressing towards a Monolith architecture
- **Large granularity of updates.** If the in-vehicle system consisted of large microservices, it would reduce the agility of software delivery into the vehicle, because larger software packages take longer to download and install
- **Performance impact.** Isolation between different microservices forces them to communicate over network channels like REST APIs, even when residing on the same physical compute node. This introduces additional delays in processing: Establishing connections (especially high impact in case of secure connections like HTTPS or TLS), serialization and deserialization of requests and responses, going through the network layers, etc.

Solution? Less isolation

So, how can we get rid of this unwanted microservice overhead? Well, let's use the same trick that helped containers win over VMs — lowering the isolation level. This makes great sense because our in-vehicle solution will be mostly composed of trusted first-party software components that don’t need to be protected from each other.

Of course, there may be scenarios that require stronger isolation, for example when you need to run untrusted “guest code.” In situations like this, you can always isolate untrusted code in a container and use that isolation to your benefit.

The example above demonstrates that containers can serve different purposes: They can be used as a packaging format, as an architectural construct, or as a hosting mechanism. These aspects are somewhat orthogonal and should be considered separately. In this example, the fact that we decided to isolate some components in a container doesn’t force us to use container-level granularity for the entire microservice architecture.

Microservice granularity

So how do we determine the right granularity for our in-vehicle microservices? Microservice architecture does not give a definitive answer; it only hints that services should be small — hence the "micro" in the name — but doesn’t tell us exactly how small.

Throughout the history of microservices, various approaches to granularity have been considered: from large quasi-monolith services to very fine-grained designs where every class is a microservice. As it's the high overhead that pushes microservices in the direction of large granularity, low overhead is associated with fine-grained microservices — and this is exactly what we're after.

Let's try to estimate how many microservices a vehicle could have in this fine-grained approach. An average modern car has over 100 million lines of software code in it. As a point of reference, the previous generation of Ford F-150 had 150 million lines of code, and that was back in 2016. Let's conservatively assume that 10% of that code becomes microservices — that's 10 million lines. With our "every class is a microservice" approach in mind, how many classes would that be? For this rough estimation, let's assume short, well-refactored classes of about 100 lines each — that gives us roughly 100 thousand microservices. Nobody would even consider running this many docker containers on the in-vehicle hardware that typically has a few gigabytes of memory, but other microservice platforms can support such scale with ease.

Actor frameworks to the rescue

One possible option for a low-overhead, fine-grained microservice platform is an Actor framework. This is a software framework based on the Actor model of computation theoretical foundation invented 50 years ago by Professor Carl Hewitt. Without going too deep into the theory, Actors are small encapsulated components that have only a few simple rules, such as:

- Actors have identity
- Actors can send messages to other Actors (or themselves)
- Actors can keep state

Beyond that, the Actor model doesn't prescribe anything else, such as: What language Actors are implemented in, where they are located (same process, same machine, elsewhere on the network), what hardware/OS they support, etc. And yet, the model has proven powerful enough to find success in many implementations across different domains, from telecom to gaming services, and other highly performant distributed systems.

Implementations of Actor models exist for most programming languages, such as Java, .NET, C/C++, Rust, Go, and many more. In some cases, Actor support is built into the language itself, as is the case with Erlang — the language behind the first success story of the Actor model when Ericsson used it in their telecom products. Some Actor frameworks even support multiple languages, thus enabling one of the benefits of container-based microservices we mentioned above — heterogeneous toolset, or the ability to mix and match different languages in the same solution. While we're on the topic of multi-lingual Actor frameworks, we should mention the availability of Actors for Web Assembly, which is a development toolset that is rapidly growing in popularity.

Most importantly, modern Actor frameworks have all the necessary capabilities we would expect from a microservice runtime, such as service placement and discovery, service communication, observability and monitoring, and developer productivity, among others.

SDV Actors: In search of a better microservices environment for software-defined vehicles | luxoft.com

Our findings

We have built several implementations of in-vehicle systems using different Actor frameworks, including a recent project based on Microsoft Orleans – an open-source framework that was developed at Microsoft Research and saw its first commercial use in 2012 during the development of Microsoft's biggest Xbox game: Halo. Orleans is famous for introducing a new Actor concept: Virtual Actors, and it also has other innovative features that we found useful in our implementation, such as Orleans Streams.

One may argue against Orleans as the best choice of Actor framework for some in-vehicle scenarios because it is built on .NET, which is a garbage-collected environment and therefore incurs additional performance overhead. However, this should not be viewed as a dealbreaker — there are successful examples of top auto manufacturers using garbage-collected software platforms in their vehicles.

Here are a few noteworthy highlights from our project:

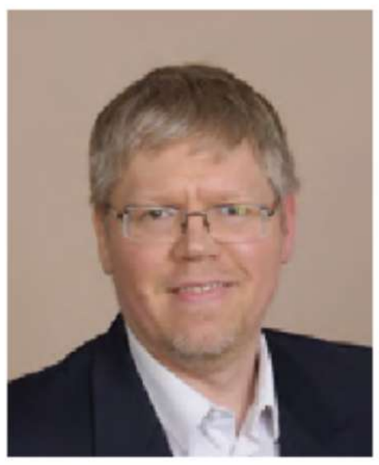
- **SDV plug-n-play**. Software-defined vehicles need to have dynamic behavior in order to react to changing operating conditions, including changes in the vehicle hardware that can happen during normal operation. A canonical example of such a hardware change is called the Trailer scenario — this is when a trailer is connected to the vehicle and the SDV system must adjust its behavior in response to this change. We have implemented the plug-n-play model for SDV Actors that allows the system to re-configure the Actor components on the fly, without requiring a system restart. After the plug-in event, the trailer's door locks started to participate in the “Central Lock” feature of the vehicle. Also, the trailer's backup camera substituted the vehicle's rear camera (which was now blocked by the trailer) on the 360-view display. Once the trailer was unplugged, original vehicle behavior was restored.

  The Actor components responsible for the trailer behavior can be provided by the trailer itself or can be obtained from a cloud repository of SDV plug-n-play components. In a situation where the plug-n-play components originate from an untrusted source, they will be executed in a separate container, isolated from the rest of the SDV Actors. This example demonstrates how fine-grained Actors and strongly isolated containers can work together to enable desirable vehicle capabilities
- **Performance.** The latest release of the Orleans framework can process hundreds of thousands of events per second on a single-node modern hardware (according to benchmarks) and we have observed similar numbers in our measurements. As a matter of example, that would be sufficient to simultaneously process a few thousand vehicle signals, each arriving at 10-millisecond intervals
- **Scale.** We were able to host 1 million Actor instances in a 4 GB of RAM footprint. Such scale should provide ample room for fairly complex microservice implementations

Conclusion

> In-vehicle microservices architectures can unlock highly desirable agility benefits in software-defined vehicles. However, the ubiquitous approach to microservices that is based on container orchestration is not efficient enough given the constraints of the in-vehicle compute environment. As a recognized innovator in software-defined vehicles, Luxoft is pursuing new approaches to implementing in-vehicle microservices, such as SDV Actors. Can your SDV solutions benefit from taking a fresh perspective on microservices? You can stay informed about this and other topics related to SDVs here or by following us on Linkedin.

About **the author**



Andre Podnozov
Chief Architect
apodnozov@dxc.com

As a seasoned technology leader at Luxoft, Andre has a wealth of expertise regarding the future of mobility. In his role as Chief Architect of Connected Mobility, he's shaping the landscape of software-defined vehicles by leveraging the latest advancements in IoT, AI/ML, Digital Twins and other exponential technologies. By bridging the gap between cloud, edge and in-vehicle systems, Andre is revolutionizing the way we think about transportation.

About Luxoft

Luxoft, a DXC Technology Company delivers digital advantage for software-defined organizations, leveraging domain knowledge and software engineering capabilities. We use our industry-specific expertise and extensive partnership network to engineer innovative products and services that generate value and shape the future of industries.

For more information, please visit **luxoft.com**