How the SOAFEE Architecture Brings A Cloud-Native Approach To Mixed Critical Automotive Systems

Matt Spencer, Principal Software Architect at Arm

A Project Cassini Reference Implementation

White Paper

As the capabilities and features deployed to modern automotive platforms is increasing at an exponential pace with the addition of features such as Advanced Drivers Assistance Systems (ADAS), Autonomous Driving (AD) and enhanced In-Vehicle Infotainment (IVI), automotive manufacturers are looking at a transition to a software-defined future.

This transition is key to the future of the market, providing extensive opportunities for margin enhancement through cost saving as well as new revenue opportunities. The rising costs of development and integration can be managed by enabling the consolidation of functional blocks within the car. Re-use of code between vehicle models and generations will help amortise the initial cost of investment in software.

The historical problem of developing code for embedded systems like those in the automotive domain has been that the software is written on the supplied API's available in the BSP that is delivered with the selected processor. There is no guarantee of portability of the application code from one processor to another due to its dependence on specific API's in the underlying BSP.

This paper introduces a solution to the problem of software portability and composability being delivered by Arm And leading technology partners in the Automotive industry.

Introduction

When we think about delivering complex software solutions made up of many functional components in a secure and managed way, we think of the large-scale applications that are running in the cloud. The infrastructure market and Cloud Service Providers (CSP's) have addressed the question of complex software deployments by adopting best "cloud-native" practices in software development and building workflows and tooling that help constrain the complexity and improve quality.

Cloud-native defines a number of technologies, workflows and design patterns that should be adopted in order to manage the complexities of developing, deploying and updating the applications live in production.

The aim of the SOAFEE project is to bring the benefits of a cloud-native development environment to address the specific challenges and constraints of the automotive domain such as Functional Safety (FuSa) and fast and precise Real time control.

One of the fundamental requirements in cloud-native is being able to decouple software from hardware. It should be possible to ensure that a workload can be easily deployed to different hardware without needing to fundamentally re-architect the underlying software. The ideal solution would be to enable binary portability without needing to recompile application code.

How Cloud-Native Applies to Automotive

The Cloud-Native Computing Foundation (CNCF) is an open source foundation that manages the specification and implementation of a number of the tools used in a Cloud-Native deployment. The following definition of Cloud-Native is owned and agreed by the members of the community:

"Cloud-Native technologies empower organizations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds. Containers, service meshes, microservices, immutable infrastructure, and declarative APIs exemplify this approach.

These techniques enable loosely coupled systems that are resilient, manageable, and observable. Combined with robust automation, they allow engineers to make high-impact changes frequently and predictably with minimal toil.

The Cloud-Native Computing Foundation seeks to drive adoption of this paradigm by fostering and sustaining an ecosystem of open source, vendor-neutral projects. We democratize state-of-the-art patterns to make these innovations accessible for everyone."

It can be seen from this definition that there is no mandate for cloud-native solutions to be deployed to the cloud. Instead, they are encouraging the use of technologies such as containers, microservices and declarative API's that build upon state-of-the-art design patterns.

These goals align very closely with our objectives in the automotive domain.
The next few sections of this paper will explain some of the technologies in more detail and how they relate to the SOAFEE objectives.

SOAFEE Uses OCI Compliant Containers

The definition of a container as prescribed by Google: *"Containers are lightweight packages of your application code together with dependencies such as specific versions of programming language runtimes and libraries required to run your software services."*

So, at a basic level, a container is a convenient way to package and deploy your application. The container environment is defined by the Open Container Initiative (OCI) and is made up of two main parts, with a third expressing standards for communicating with a container registry (such as hub.docke r.com)

- Container Runtime Specification
- Container Image Specification
- Container Distribution Specification

The runtime specification covers the system requirements and interfaces that need to be made available in order to allow a container to operate successfully on your system. Whilst the image specification defines how the images should be composed for them to be acceptable to the container runtime.

The power of this standards-based approach to the container ecosystem is that it encourages people to innovate around the container runtime implementation to create a solution that meets the domain specific requirements of a particular deployment. And whilst the OCI have the reference implementation of a container runtime with runc, there are many other container runtimes which apply to different domains.

A non-exhaustive list of available container runtimes includes:

Runtime	Characteristics
runc	Reference runtime from OCI, implemented in go-lang
crun	Small footprint lightweight runtime implemented in C
gvisor	Sandboxed runtime with increased security by limiting system API access
kata	Virtualized runtime that leverages KVM for better security and separation
runx	Virtualized runtime built on Xen
...	Other container runtimes are available

The point to be made here is that there is not a one-size-fits-all solution to which container runtime you should use. But you are guaranteed that any container built to the OCI container specification will work unmodified on your platform.

SOAFEE aims to make use of this powerful separation between application and runtime to ensure that a container runtime exists that meets the demanding deployment characteristics of the automotive domain, and that the upstream standards within the OCI are able to express these requirements where needed.

Microservice

The Microservice Architecture is a software design pattern that enforces the creation of loosely coupled, collaborating services that can create a functional solution by composing the services together. These services have a clearly defined inbound and outbound interfaces, which creates the contract the service has with other components in the system.

In a cloud-native deployment, the microservice would be encapsulated in a container. This enables it to execute in the defined container runtime environment, and deployment to be managed and monitored by an orchestrator - more on this later.

Microservices are defined as loosely coupled because changes to one service should not impact the performance of another service in the system so long as the expected behaviour as defined in the contract of the inbound and outbound API's are honoured. This characteristic also means that the microservice can be tested in isolation from the rest of the system, which enables the larger complex system to be decomposed into unit testing for the individual service before going to integration testing of the fully composed system.

Orchestrator

The Orchestrator is a vital component of the cloud-native system. It manages the configuration, deployment and monitoring of the microservice based solution. The orchestrator itself is composed of several standard interfaces:

Interface	Description
CRI	The interface between the orchestrator and the container runtime
CNI	Container Network Interface, a standard mechanism for configuring and controlling network, firewall, etc.
CSI	Container Storage Interface, how to expose storage available to your container instances
Device Plugin	Enables managed access to system resource within containers such as /dev/video0
...	There are other standards that may be applicable

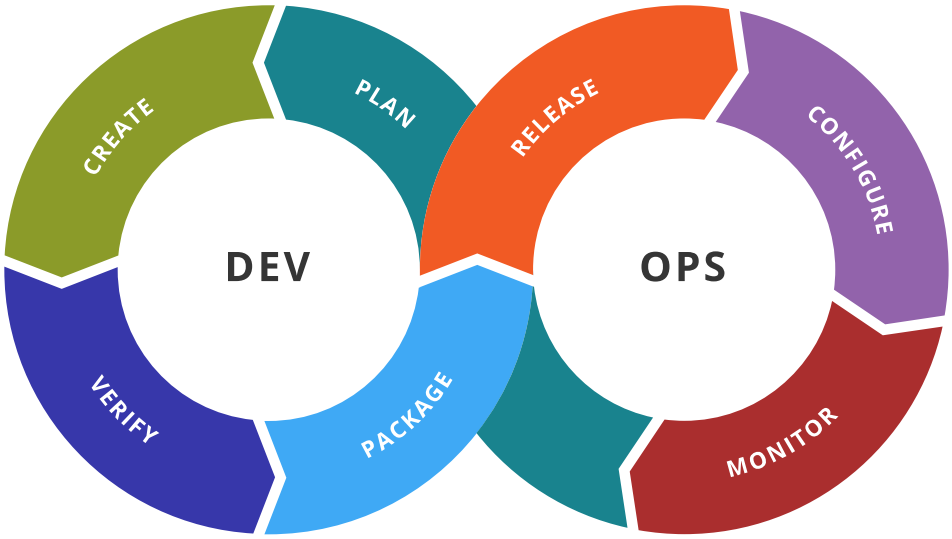
As with all aspects of the cloud-native ecosystem, there are multiple implementations of these standard interfaces that meet use-case specific needs. If you want your container runtime to be managed by the orchestrator, it needs to implement the CRI interface, all of the runtimes described earlier in this document comply with that requirement.

When these interfaces are bought together under the control of the orchestrator, it enables complex application deployments to be managed by configuring the network to enable communication between the microservices and access to data sources that are needed for them to function properly.

There are a number of options for your orchestrator, the default being kubernetes (aka k8s) but there are implementations with smaller footprints such as k3s which are more suited to embedded and resource constrained environments.

DevOps

The workflow aspect of the cloud-native is often referred to as a DevOps process.



Source: commons.wikimedia.org/wiki/File:Devops-toolchain.svg

The workflow is split into two main parts. Dev covers the development workflow, and Ops covers the operations aspect of the deployment. By bringing these two disciplines together in a clearly defined and managed way, it is possible to streamline the development, deployment and continual improvement of the applications managed under this workflow.

You can see from the image that this is a continual process where the output of monitoring the operation of the container in deployment feeds back into the next phase development cycle.

This is how continual improvement of the quality of the delivered workloads is enabled. Over time, adoption of this process will increase overall quality whilst reducing time to market and overall cost.

How SOAFEE Extends Cloud-Native

The SOAFEE project leverages cloud-native framework in order to benefit from the best practices and standards that are used, the problem is that there are additional requirements and constraints that come into scope when working with Automotive solutions. These include the ability to deploy workloads to heterogeneous compute architectures with a mixture of application processors and real time processors with an array of accelerators available.

Through the SOAFEE Working Groups, SOAFEE aims to understand the current gaps in cloud-native implementations and work upstream within the relevant standards bodies. Collaborative effort to close those gaps will enable cloud-native solutions to be applicable in the automotive and safety relevant domains.

Orchestrator Enhancements – For Safety and Real Time

There are some aspects of the orchestrator scheduling framework that can cope with these additional requirements today with use of standard techniques such as:

- Node affinity to constrain where a workload will be deployed
- Taints and Tolerations to describe how certain nodes attract or repel each other
- Pod overhead to describe the amount of system resources that will be consumed by a particular workload

But there are gaps in this system as it stands today, and SOAFEE will work upstream with the standards community to normalize the language by which real time and safety requirements are described.

Examples – Real Time Requirements

- Required I/O bandwidth
- Guaranteed execution time
- Cache policies

Examples - Safety Requirements

- Freedom from interference
- Split-lock core
- Availability

These areas of extension will be worked on within the respective SOAFEE technical working groups to come up with a common language that can be used to express intent to the Orchestrator. This will enable workloads to be deployed to the correct part of the underlying system and low-level architectural features to be configured for each Pod to guarantee meeting those requirements.

Container Runtime Enhancements

Now that the orchestrator is able to express properly the additional runtime requirements of our workloads, work will be done to enhance the container runtime to enable it to meet these needs. The prime path proposal is to use a virtualized container runtime such as runx mentioned earlier, to enable use of the VMM in collaboration with the runtime itself to separate the control of privileged system resources through the VMM from the lower privileged execution environment of the container runtime.

The SOAFEE working groups will be responsible for selecting the correct initial execution environment and then working upstream with the OCI standards body when modifications are needed, but also with the chosen container runtime to implement the enhancements.

Portable Workloads

A key SOAFEE value proposition is the reuse of workloads. This is how we enable reducing the cost of deployment for these complex software solutions by enabling the specific microservices that compose the final application to be reused across different product lines and solutions without modification.

In order to achieve this, we need to understand how we can give consistent access to accelerators and high bandwidth IO devices to the workloads without needing them to be recompiled against an architecture specific implementation of the accelerator or IO device. We have to be aware that in exploring this concept, we need to keep in mind the domain specific requirements around Functional Safety and Realtime.

One industry standard that comes to mind is VirtIO. VirtIO providers para-virtualised access to accelerators which effectively standardises the workloads view of the accelerator whilst enabling efficient offload to a backend accelerator.

On the surface, VirtIO seems like the perfect solution to the portability question, but there are problems with the current release of VirtIO. Firstly, it has not been designed with Functional Safety or Realtime workloads in mind. Secondly, the interfaces do not cover all of the requirements of the Automotive domain. For example, there is no VirtIO interface for Machine Learning acceleration through a userspace like TVM, and there are no standard interfaces to Automotive specific IO devices like the CAN bus.

As SOAFEE prioritises adoption of pre-existing standards and adapts them to be fit-for-purpose in a functionally safe domain, rather than try and create a new standard which could drive ecosystem fragmentation, SOAFEE will work upstream within the VirtIO standards body to resolve the issues. This would include work items around ensuring we can express realtime constraints on virtqueues, and that there is industry collaboration to define sensible VirtIO implementations for the missing interfaces.

Testing and Validation

One great opportunity that comes from creating a microservices based solution is that we enable tooling for CI/CD for component and system level testing and validation. For example, we can make use of the workload portability features of SOAFEE to allow execution of workload training and testing in the cloud to give a first level confidence in the workloads performance. This same workload can then be deployed to a lab-based infrastructure to enable application of both software and hardware in the loop validation.

The standard DevOps workflow introduced earlier outlines how the cloud-native deployments manage the quality of complex workloads, SOAFEE enables adoption and enhancements of this workflow. The SOAFEE project is working with System Integrators, Tool Vendors and CSP's to make this capability a reality.

There will be more details on this very important aspect of SOAFEE soon.

Open Source Reference Implementation

All of this effort will come together in the form of an open source reference implementation of the SOAFEE requirements, including all the necessary components to realise the cloud-native vision of the project. The reference will be delivered in the form of Yocto recipes that will enable porting the base platform to alternate hardware.

This reference implementation can then be used by upstream rich software stacks such as Autoware, AGL, and others to implement truly portable, containerized, microservice based implementations that will run on any platform that implements the SOAFEE architecture.

Details of how to build the current SOAFEE stack and the supported hardware is available on the SOAFEE project page. Details can be found at gitlab.arm.com/soafee.

Conclusion

The SOAFEE project brought together automakers, semiconductor and cloud technology leaders to define a new open-standards based architecture for the software-defined vehicle. It delivers a reference implementation which enables cloud concepts like container orchestration to be combined with automotive functional safety for the first time. It builds on the successes of the Arm initiative Project Cassini, which defines standard boot and security requirements for Arm architecture.

By embracing cloud-native technologies, we enable advanced CI/CD techniques such as software and hardware in the loop, and the use of cloud-based infrastructure for training and validation.

With SOAFEE, we can reduce the complexity of the software-defined vehicle whilst also reducing the cost of development and deployment. It will also enable maximum reuse of the ecosystems investment in software by enabling deployment of existing workloads to new architectures without the need to re-integrate. For more information, please visit our website.

Glossary of Terms

Term	Meaning
AD	Autonomous Drive
ADAS	Advanced Driver-Assistance Systems
AGL	Automotive Grade Linux
API	Application Programming Interface
BSP	Board Support Package
CI/CD	Continunal Integration / Continual Deployment
CNCF	Cloud-Native Computing Foundation
CSP	Cloud Service Providers
IVI	In Vehicle Infotainment
OCI	Open Container Initiative