Eclipse SDV

The Eclipse SDV Compendium

Open source technologies for software-defined vehicles



Table of contents

Editorial 3

Reference Architecture & Example Deployment

Eclipse SDV Blueprints 9

Orchestration

The Eclipse SDV orchestration landscape 19

Eclipse Ankaios 24

Communication Layer

An SDV communication layer with Eclipse Zenoh and Cyclone DDS 29

Why Eclipse uProtocol matters 34

Eclipse Kuksa as a standardised vehicle abstraction layer 39

Vehicle Software & Application Layer

How Eclipse Velocitas accelerates SDVs 43

On the necessity of an application framework for SDV projects 48

Middleware & Platform Services

Eclipse S-CORE: The open source middleware foundation for the SDV 54

Low-Level Software & Hardware Abstraction

Eclipse ThreadX 57

Silicon & Hardware Platform Layer

Open source silicon: The new frontier of software-hardware co-design 62

Cloud, Tooling & Lifecycle Management

An open development toolchain to support SDV stack development 67

Eclipse Capella and SysOn: MBSE for SDVs 73

Safety, Security & Compliance

Functional Safety in the era of open source 81

Outlook & Future Evolution / AI

SDVs in the age of AI: The rise of Skynet 86

Conclusion & Outlook 90

The Authors 92

Editorial

From individual projects to an open and collaborative SDV architecture

BY CHRISTIAN HEISSENBERGER & SARA GALLIAN

A software-defined vehicle (SDV) is not simply a car with more software. Vehicles have depended on software for decades. The change is more fundamental: software is becoming the main means by which vehicle functions are created, integrated, improved, and operated throughout the life of the product. Features can be developed against stable interfaces, deployed to different computing platforms, updated after production, and connected to services beyond the vehicle.

Several developments drive this shift at once. Electrification is changing powertrain and energy management, while advanced driver assistance and automated driving functions require substantial computing power and large data flows. Vehicle E/E architectures are moving towards central computing and zonal I/O. Cybersecurity, software update regulation, functional safety, and long product lifecycles also demand control over versions, dependencies, and evidence. The industry must deliver more software faster while preserving vehicle reliability.

The difficult part is making these technologies work together. A vehicle contains microcontrollers with strict timing requirements, real-time operating systems, high-performance computers, communication middleware, data abstraction, applications, cloud services, engineering tools, and fleet operations. They are developed by different organisations and follow different release cycles. Repeating their integration for every vehicle program makes reuse unnecessarily hard.

This Compendium presents a practical view of open source technologies. It does not prescribe one implementation or claim that one stack can replace every established automotive platform. Instead, it shows how specialised projects can form a modular system when their responsibilities and interfaces are clear. Chapter topics range from runnable Eclipse SDV Blueprints and orchestration to communication, vehicle data abstraction, application development, middleware, real-time systems, open silicon, engineering toolchains, model-based systems engineering, functional safety, and AI.

Open collaboration builds better SDVs

A large share of vehicle software is essential, but not differentiating. Every organisation needs to build infrastructure, communication mechanisms, deployment tools, diagnostics, data access, and test environments. Rebuilding these foundations separately consumes engineering capacity without creating a clear advantage for the customer.

Open source offers a way to share this effort. OEMs, suppliers, semiconductor companies, software vendors, and research organisations can collaborate on common foundations while continuing to compete through products, algorithms, services, integration quality, and vehicle features.

However, access to source code alone is not enough. As we have explained in a blog[1] on Android Automotive OS and SDV governance, a project can be publicly available while one company still controls its roadmap, contribution process, licensing, or surrounding services. For vehicle platforms that must be maintained for many years, such dependencies can become a strategic risk.

Thomas Froment makes a similar point in his discussion of trust in development tools. A product may have an open core while important extensions, binaries, integrations, or services remain proprietary. It is therefore necessary to consider the complete dependency chain, not only the licence of a single repository.[2]

Open collaboration goes further than publishing code. It combines open licences with transparent development, fair participation, shared intellectual property rules, and vendor-neutral governance. It should also give users the practical ability to build, operate, maintain, and further develop the technology without depending on one organisation. This supports long-term maintenance, reduces vendor dependency, and allows knowledge to be shared across the ecosystem.

Openness does not remove engineering responsibility. An open source component is not automatically secure, functionally safe, production-ready, or certified. It still requires a defined operating context, qualified integration, verification, vulnerability management, and clear lifecycle ownership. Its advantage is that the implementation, interfaces, limitations, and development history can be inspected and improved by a broader community.

The Eclipse SDV community follows a code-first, but not code-only, approach. Working software is developed together with requirements, architecture, interfaces, tests, processes, traceability, and operational evidence. Eclipse SDV projects bring these elements together, reveal integration gaps, and feed improvements back into the ecosystem. In this Compendium, the projects therefore show how an open SDV architecture can be specified, implemented, tested, and operated in practice.

[1] https://eclipsesdv.org/blogs/googles-aaos-sdv-open-source-and-the-open-question-of-governance/

[2] https://blogs.eclipse.org/post/thomas-froment/when-%E2%80%9Copen%E2%80%9D-isn%E2%80%99t-open-enough-rethinking-trust-dev-tools

The Eclipse SDV Landscape as the common map

The Eclipse SDV Landscape[3] (**figure 1**) provides the navigation model for this Compendium. It is an overview of Eclipse projects relevant to software-defined vehicles, generated from project data and organised through a maintained category structure. It is a map of capabilities, not a prescribed bill of materials. Real systems will combine projects from several categories.

The Landscape groups the ecosystem into seven main areas: AI; Tooling & DevOps; Service Fabric and Communication; Runtime and Orchestration; Distributions and Frameworks; RTOS; and Example Applications. Tooling & DevOps covers design, development, testing and validation, deployment, and process frameworks. Service Fabric and Communication includes middleware, communication protocols, service mesh technologies, and digital twins. These categories guide the overview below.

The Landscape focuses on software projects. The Compendium adds the hardware and E/E context below that map, because processors, microcontrollers, memory, and networks shape timing, isolation, energy use, and cost. Safety, security, compliance, and lifecycle management cut across every category. The architecture is not a simple tower of layers, but a set of collaboration capabilities across the vehicle-to-cloud continuum.

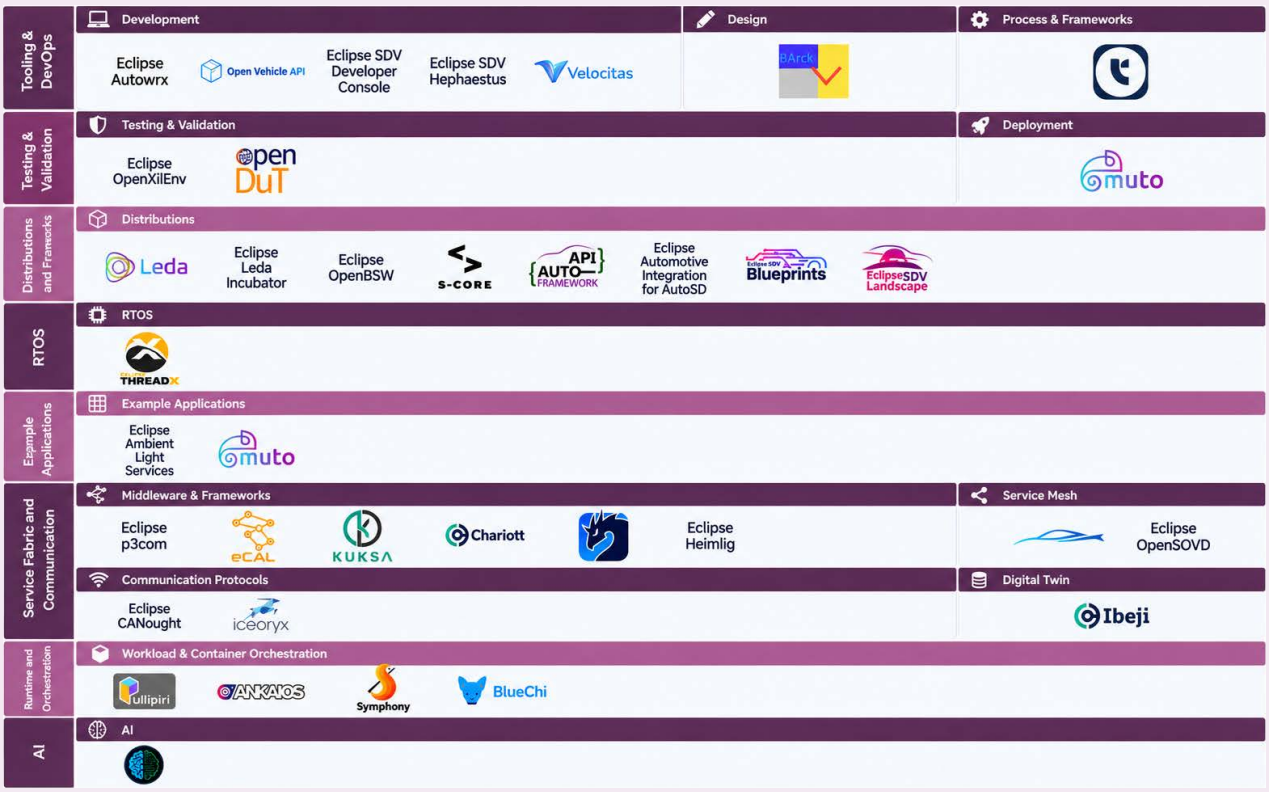


Figure 1: Eclipse SDV Landscape in September 2026

[3] https://projects.eclipse.org/projects/automotive.sdv-landscape

Architectural overview

At the physical foundation are the vehicle's compute and I/O resources. Future architectures will mix central computers, domain and zonal controllers, and deeply embedded microcontrollers. Centralisation can reduce duplication and make functions easier to update, but it does not mean every task moves to one large computer. Fast control loops and direct sensor or actuator access may remain close to the edge. A zonal controller may act mainly as a gateway, or it may also execute local real-time functions.

Above the hardware, **RTOS and base platform software** provide execution environments for time-critical and embedded functions. Deterministic scheduling, bounded latency, small memory footprints, and reliable startup behaviour matter here. General-purpose operating systems on high-performance computers support richer applications, isolation, containers, networking, and modern workflows. Mixed-criticality vehicles will use both, with clear boundaries and controlled communication. The Landscape describes RTOS technologies as a foundation for low-latency functions, while distributions and frameworks combine operating systems, middleware, and management components into usable platforms.

Distributions and Frameworks assemble lower-level capabilities into a coherent starting point. They can provide operating-system integration, middleware choices, reference configurations, and development conventions. They reduce repeated platform work while remaining modular enough to exchange components when required.

Runtime and Orchestration controls what software runs, where it runs, and how its state changes. Software lifecycle management cannot end when an image is flashed at the factory. Workloads may need to be started for diagnostics, activated for a market or customer, replaced after an update, or rolled back when health checks fail. Vehicle-side orchestrators manage execution on ECUs and high-performance computers; fleet-level systems coordinate campaigns across many vehicles. Different criticality levels may require complementary tools rather than one universal orchestrator.

Service Fabric and Communication connects these execution environments. It includes protocols, middleware, service interaction, vehicle-data abstraction, and digital-twin concepts. The central principle is decoupling: applications should depend on stable data models and service contracts rather than on a particular ECU address, bus frame, or transport. Eclipse Zenoh, Eclipse Cyclone DDS, Eclipse uProtocol, Eclipse Kuksa, and related middleware address different parts of this problem. A perception stream, a vehicle-service request, cloud telemetry, and a hard real-time control path have different requirements and should not be forced through the same mechanism.

At the **application level**, frameworks and SDKs help engineers use those abstractions. Applications can consume standardised vehicle data and call services without

carrying the details of every vehicle variant. When hardware, operating systems, or middleware change, the application should remain stable wherever timing and safety constraints permit. Eclipse Velocitas, the Eclipse Automotive API Framework, Eclipse Kuksa, and Eclipse S-CORE-related approaches illustrate different parts of this area.

AI is shown as its own Landscape category, but appears across the architecture. It can run in engineering tools, cloud services, vehicle computers, or embedded accelerators. Its placement must be driven by latency, energy use, privacy, safety, and the ability to supervise its behaviour. AI does not replace deterministic software.

Tooling & DevOps spans the architecture rather than sitting above it as a final layer. Architecture models, source code, build configurations, test assets, deployment manifests, safety evidence, and operational feedback must evolve together. Simulation, continuous integration, reproducible environments, traceability, and secure deployment are all part of the product lifecycle.

Finally, **Example Applications** and **Eclipse SDV Blueprints** show whether the pieces can form a system. A diagram can suggest compatibility; a runnable example proves that components can be built together, configured, started, observed, and changed. The Blueprints in this Compendium cover vehicle signals, service-to-signal translation, fleet management, software orchestration, and cloud-connected applications. Integration findings can be fixed in the open, improving components for the next adopter.

SDV levels of maturity

Understanding the available building blocks is only one part of the picture. It is also useful to describe how far a vehicle has progressed in becoming software-defined. The project Eclipse SDV-LVL[4] (Software-Defined Vehicle Levels of Value & Lifecycle) addresses this question by developing a vendor-neutral framework for assessing SDV maturity. The project aims to establish a common taxonomy for classifying technical capabilities and their impact on the customer experience, while also considering the technical and organisational enablers needed to achieve them. Similar to the SAE levels that created a shared language for driving automation, SDV-LVL is intended to make discussions about software-defined vehicles more precise and comparable. Although the taxonomy has not yet been implemented in the Eclipse SDV ecosystem, it will complement the Eclipse SDV Landscape once established: while the Landscape maps the technologies and capabilities available for building an SDV, SDV-LVL will provide a common language for describing the level of software-defined capability that a vehicle has achieved.

[4] https://github.com/eclipse-sdv-lvl/sdv-lvl

How to read this Compendium

The chapters can be read individually, but gain value in combination. Eclipse SDV Blueprints show integrated use cases. The orchestration chapters explain how software is deployed and controlled. Communication, service mesh, and vehicle-data chapters describe how components interact. Application frameworks and middleware support portable vehicle functions. RTOS and open silicon connect the software architecture to deterministic execution and hardware. Tooling and MBSE address how the system is designed and maintained. The final chapters examine functional safety and AI.

The common thread is the move from closed, program-specific integration towards reusable, inspectable, and composable foundations. The Eclipse SDV Landscape keeps the ecosystem understandable; projects provide the building blocks; and Blueprints show how they become working systems. This Compendium invites readers to examine the parts, question the boundaries, run the examples, and contribute where the architecture needs to mature.

But the SDV Compendium is not complete. There are many additional initiatives, some of which are not yet described in detail in this version.



Eclipse SDV Blueprints

Realising the SDV through runnable designs

BY NACI DAI

The transition of the global automotive sector toward SDVs requires a fundamental shift from isolated, proprietary hardware-software silos to shared, interoperable, and open source software foundations. While high-level architectural standards are essential, their actual utility in reducing engineering duplication depends heavily on the availability of concrete, runnable, and easily reproducible implementations. This operational imperative is addressed directly by the Eclipse SDV Blueprints initiative, a collaborative effort designed to bridge the gap between abstract design patterns and real-world deployment.[5] Rather than generating entirely greenfield systems, the initiative acts as an integration hub, combining existing components from the broader Eclipse SDV project portfolio to build composable, executable showcases. In doing so, it lowers barriers to entry, accelerates industry-wide experimentation, and establishes a critical feedback loop that enhances the maturity of individual projects.

The integration testbed: collaborative feedback

Beyond acting as educational reference designs, blueprints function as living integration testbeds. Within the highly decentralised Eclipse SDV landscape, individual projects like Eclipse Kuksa, Eclipse Velocitas, and Eclipse uProtocol are often developed independently, each targeting a specific layer of the vehicle software stack. Bringing these projects together within a single blueprint exposes real-world interoperability challenges, creating a continuous feedback loop.

When integration issues emerge, the blueprint project serves as the debugging environment. For example, during the integration of the Fleet Management Blueprint with Eclipse Hono, developers identified and fixed a critical edge-case crash where the telemetry consumer failed during startup.

[5] https://github.com/eclipse-sdv-blueprints/blueprints
Website: https://github.com/eclipse-sdv-blueprints/blueprints-website
Overview | Eclipse SDV Blueprints: https://sdv-blueprints.eclipse.dev/docs/overview/

Identifying these configuration gaps and software defects directly within the blueprint environment ensures that the fixes are propagated back to the parent projects, thereby enhancing the overall security and stability of the entire SDV ecosystem.

Blueprint implementations

To understand how the Eclipse SDV working group achieves modular interoperability, it is necessary to analyse the technical designs of the primary blueprints. The following table provides a brief overview of the existing blueprints:

Blueprint Name	Primary Target & Use Case	Core Eclipse Projects Integrated	Protocols & Transports	Hardware / Runtime Environment
Companion Application	Vehicle state control (e.g., seat adjustment based on driver profiles)	Velocitas, Kuksa.val Databroker, Leda, Kanto	MQTT, gRPC	VSCode DevContainer, Leda (QEMU, container, or hardware)
Fleet Management	Commercial truck fleet telemetry, monitoring, and analysis	Kuksa.val, uProtocol, Hono	Zenoh, MQTT, Kafka (Hono transport)	Docker Compose, Podman
Service to Signal	Implementation of standard service interfaces mapped to signals	uProtocol, Zenoh, Kuksa	Zenoh, uProtocol	Cargo / Rust Runtime
Software Orchestration	Dynamically provisioning in-vehicle workloads (e.g., Smart Trailer scenario)	Ankaios, BlueChi	gRPC, Systemd D-Bus	Podman, Systemd, Quadlets
ROS Racer	Cloud-native orchestration and OTA updates for autonomous racing	Muto (Agent, Core, Composer), Symphony	ROS 2 (CycloneDDS), MQTT	Podman/Docker containers
E2E Vehicle	Edge-to-cloud	Ankaios	CAN	Raspberry Pi
Signals	Signal pipeline and E/E hardware integration	Kuksa, S-CORE, ThreadX	(SocketCAN), MQTT, gRPC	4/5, Arduino (MCU1), CAN Bus (MCP2515)

Companion Application Blueprint

The Companion Application Blueprint showcases a complete, end-to-end flow for interacting with vehicle hardware.[6] The use case focuses on a seat adjustment service that moves the driver's seat to physical coordinates retrieved from an external driver profile database.

Architecturally, the blueprint is structured to decouple vehicle-agnostic business logic from physical hardware.[7]

The system simulates mobile triggers by publishing JSON payloads to an MQTT broker. This trigger is captured by a Seat Adjuster application developed using the Eclipse Velocitas framework (see page 42). The Seat Adjuster does not communicate with the seat motor directly; instead, it issues a gRPC update to the Kuksa Databroker, which maintains the state of the vehicle using the COVESA Vehicle Signal Specification (VSS, see page 38).

The databroker, running inside an Eclipse Leda Linux image managed by the Eclipse Kanto container runtime, notifies a hardware-specific Mock Service. This service subscribes to the VSS seat position signals, processes changes, and would – in a production vehicle – interact with the seat ECU over a physical CAN bus.

Decoupling these layers ensures that the core application logic remains highly portable across different vehicle platforms. Developers can execute the client from a container to query values or invoke MQTT publish commands directly to simulate target seat coordinate adjustments.

Fleet Management Blueprint

Designed as a close-to-real-life telematics and monitoring showcase for logistics fleet operators, this blueprint orchestrates data flows from vehicle edge nodes to cloud backends.[8] Vehicle telemetry is simulated by playing back GPS, speed, and fuel data from a CSV file via the Kuksa CSV Provider, which regularly updates the local Kuksa Databroker.

An in-vehicle Eclipse uProtocol entity, known as the FMS Forwarder, reads these signal changes and packages them into standard uProtocol notifications.

A key technical strength of this blueprint is its transport-agnostic design. The communication channel between the in-vehicle FMS Forwarder and the cloud-based FMS Consumer can be toggled via configuration:

- **Eclipse Zenoh Transport:** A low-latency, decentralised peer-to-peer transport framework that routes uProtocol payloads directly to the FMS Consumer backend.
- **Eclipse Hono Transport:** A highly scalable enterprise IoT gateway configuration that utilises Eclipse Hono's MQTT adapter to ingest vehicle data, routing it through an Apache Kafka messaging infrastructure to the consumer.

[6] https://github.com/eclipse-sdv-blueprints/companion-application

[7] https://github.com/eclipse-sdv-blueprints/companion-application/blob/main/interact-seat-adjuster.md

[8] https://github.com/eclipse-sdv-blueprints/fleet-management

Once the backend FMS Consumer receives these payloads, it writes the time-series measurements into an InfluxDB database. This database is queried in real time by an HTTP-based rFMS API server and visualised via a Grafana dashboard, providing fleet managers with direct insight into vehicle position, fuel consumption, and performance. Provisioning devices under the Eclipse Hono configuration is accomplished via a shell script that registers the vehicle credentials directly with the target sandbox infrastructure (see **figure 1**).

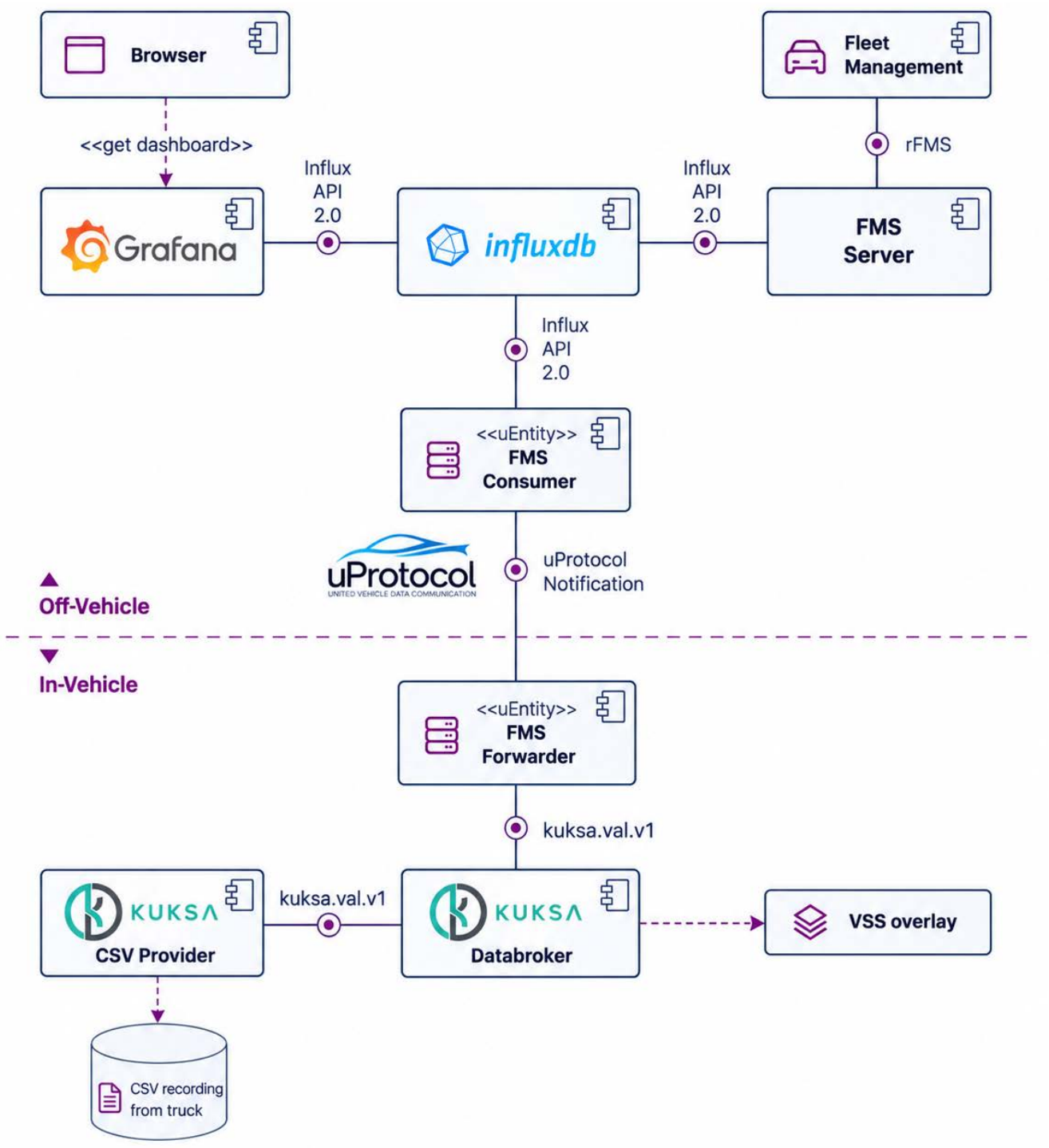


Figure 1: Fleet Management Blueprint

Service-to-Signal Blueprint

This blueprint (see **figure 2**) demonstrates the practical realisation of the Service-to-Signal design pattern, which is essential for translating service-oriented API calls into low-level hardware control signals.[9] It implements a standard COVESA uService interface for controlling a vehicle's horn. Developed in Rust, the horn service uses the Eclipse Zenoh transport protocol to establish a uProtocol communication channel. When an offboard application invokes the service-oriented Horn RPC API, the service translates this high-level method call into an explicit signal update, modifying the corresponding VSS horn-state indicator within the local Kuksa Databroker.[10]

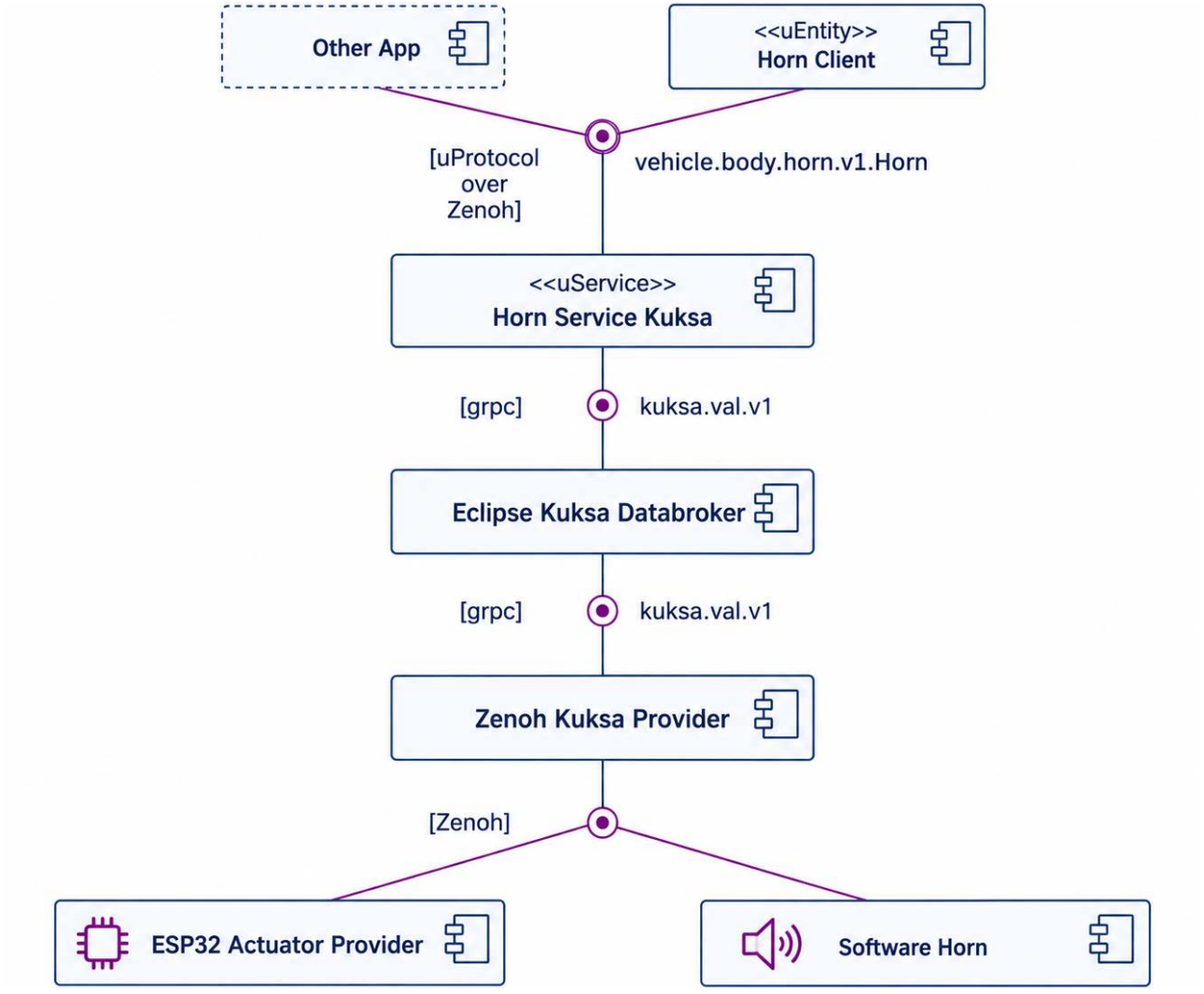


Figure 2: Service-to-Signal Blueprint

[9] GitHub – eclipse-sdv-blueprints/service-to-signal: The service-to-signal blueprint is an example to showcase how to use the Eclipse uProtocol (https://github.com/eclipse-uprotocol) to make a vehicle service available in a vehicle network and connect the service implementation with potential physical hardware: https://github.com/eclipse-sdv-blueprints/service-to-signal service-to-signal/.gitmodules at main · eclipse-sdv-blueprints/service, https://github.com/eclipse-sdv-blueprints/service-to-signal/blob/main/.gitmodules

[10] service-to-signal/components/horn-service-kuksa/README.md at main · eclipse-sdv-blueprints/service-to-signal – GitHub, https://github.com/eclipse-sdv-blueprints/service-to-signal/blob/main/components/horn-service-kuksa/README.md

Software Orchestration Blueprint

Managing complex, dynamic applications across vehicle networks requires advanced container orchestration.[11] The Software Orchestration blueprint (**figure 3**) demonstrates how to handle these challenges by comparing and integrating two distinct orchestrators:

Orchestration Parameter	Eclipse Ankaios Integration	Eclipse BlueChi Integration
Underlying Architecture	Server-Agent setup with unique centralised API	Multi-node controller tied to systemd D-Bus
Container Runtime	Podman workloads	Systemd services / Podman quadlets
Deterministic Focus	Dynamic scale workload scheduling and resilience	Deterministic control for highly regulated ecosystems

The blueprint showcases these tools in a “Smart Trailer” use case. When a physical trailer is connected to the vehicle, the system raises a signal that is routed to the orchestrator. The orchestrator then dynamically provisions the necessary smart trailer provider application, resolving dependencies via services like Eclipse Chariott for discovery, Mosquitto for messaging, and Freyja for digital twin synchronisation. This setup adjusts the vehicle’s body functions, braking profiles, and powertrain management dynamically, terminating the workload once the trailer is disconnected.

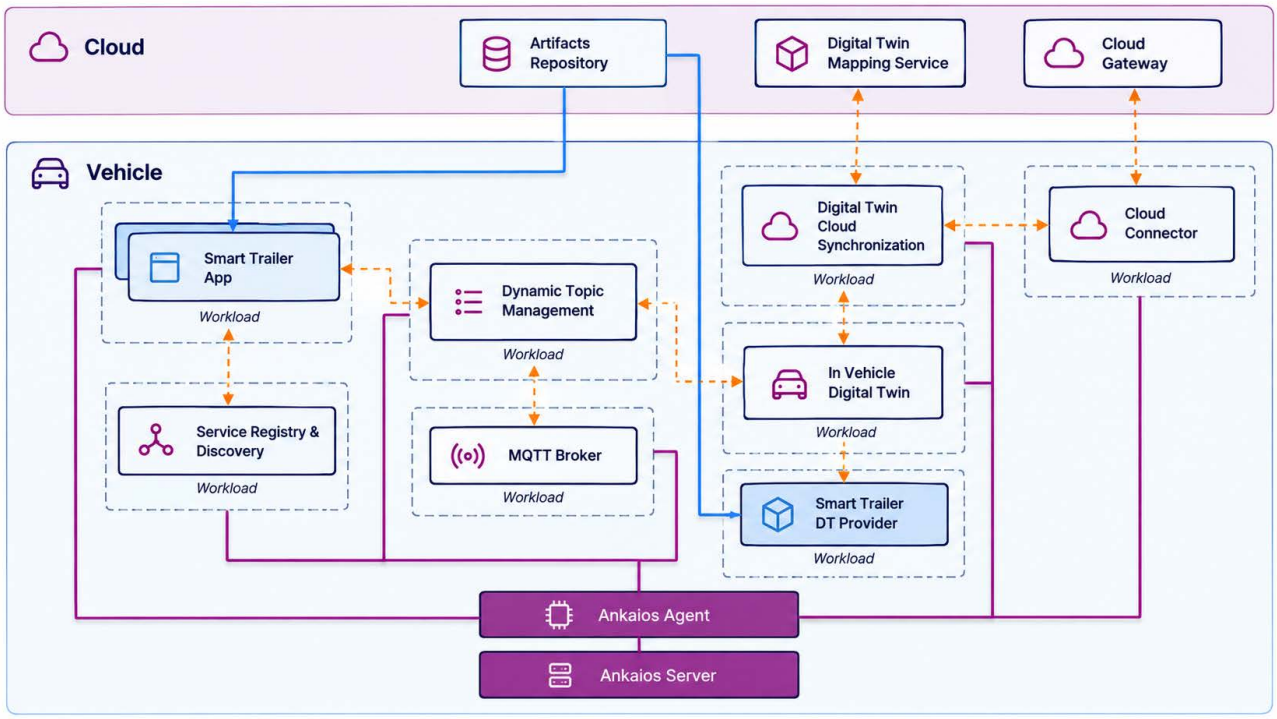


Figure 3: Orchestration Blueprint

[11] eclipse-sdv-blueprints/software-orchestration - GitHub, https://github.com/eclipse-sdv-blueprints/software-orchestration

ROS Racer Blueprint

The ROS Racer blueprint (**figure 4**) applies cloud-native orchestration with Eclipse Muto and Eclipse Symphony for over-the-air software updates to autonomous scale racing.[12] The simulation layer relies on the physical F1Tenth Gym simulator running ROS 2, visualising three vehicles on a race track. Each racecar runs an Eclipse Muto stack, representing a vehicle's edge compute node:

- **Muto Agent:** Maintains connectivity to the cloud, exposing an MQTT gateway and integrating with Eclipse Symphony for cloud-side orchestration.
- **Muto Core:** Synchronises state attributes with an Eclipse Ditto digital twin.
- **Muto Composer:** Manages the dynamic software lifecycle.

When a new driving algorithm is deployed (such as shifting from a conservative to an aggressive gap-follower approach), the cloud orchestrator (Eclipse Symphony) pushes a new stack definition. The Muto Composer downloads the tarball from an artifact server, validates, builds, resolves dependencies, and launches the new driving nodes. If the new algorithm fails or crashes during startup, the Composer automatically triggers a rollback to the last known stable driving configuration, showcasing a critical mechanism for maintaining vehicle safety during OTA updates.

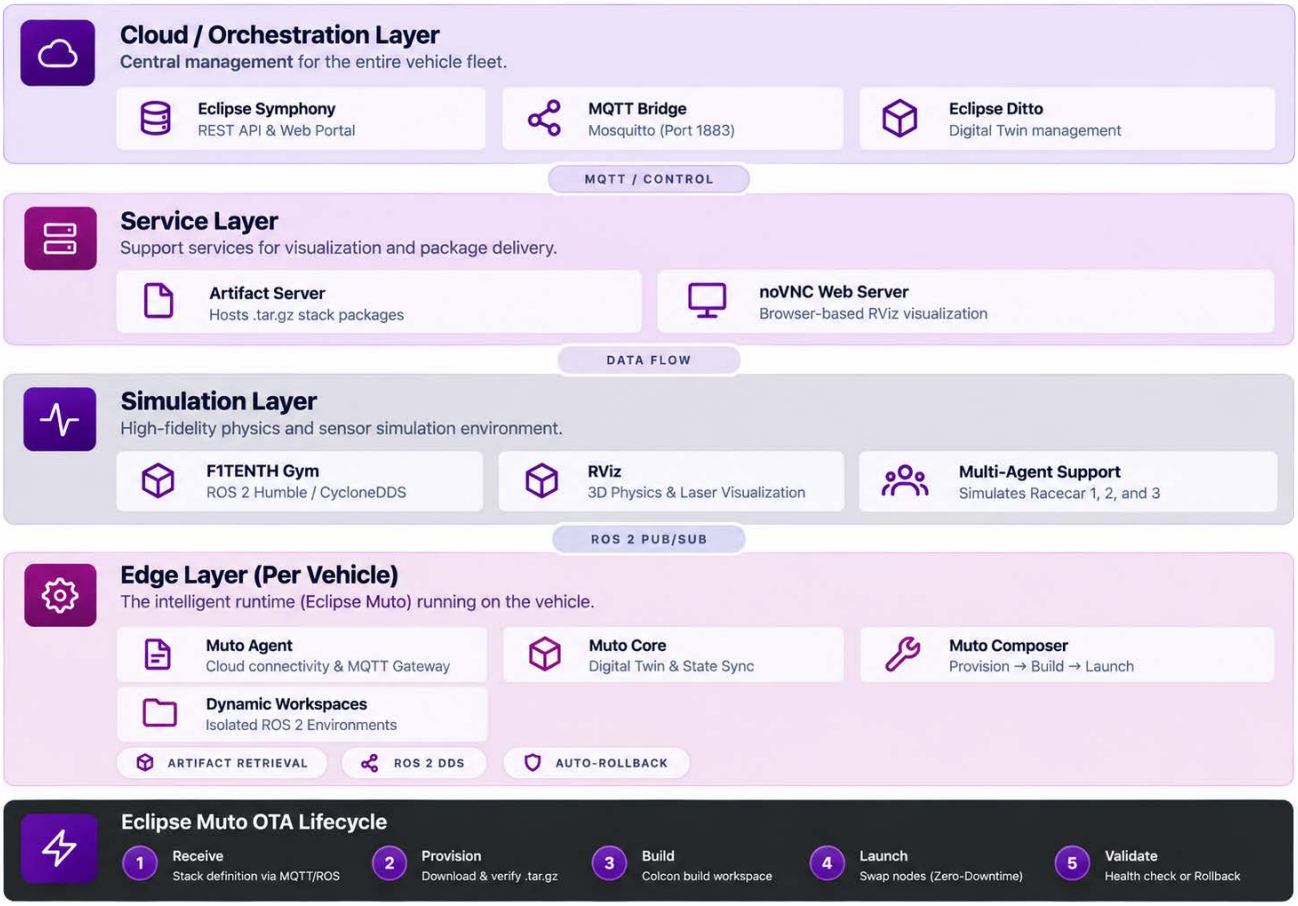


Figure 4: ROS Racer Blueprint

[12] eclipse-sdv-blueprints/ros-racer: Take ROS, F1Tenth and a race track and see where it all gets you! – GitHub, https://github.com/eclipse-sdv-blueprints/ros-racer

End-to-End Vehicle Signals Demo Blueprint

The End-to-End (E2E) Vehicle Signals blueprint (**figure 5**) provides a comprehensive demonstration of physical and virtual integration, bridging embedded microcontrollers with high-performance computing (HPC) platforms and cloud backends.[13]

The physical hardware architecture consists of a Raspberry Pi 5 acting as the primary HPC (running Ubuntu, Eclipse Ankaios and the Eclipse Kuksa Databroker), a Raspberry Pi 4 for high-level safety-critical control (utilising Eclipse S-CORE, see page 53), and an Arduino Uno (MCU1) with an MCP2515 CAN controller to drive an 8-segment LED strip. The system aligns all telematics and control data to the COVESA VSS standard, tracking left/right indicators, brake states, and driver identifiers. The signal flow proceeds through an integrated execution pipeline:

- **Input Generation:** The driver initiates input via a physical joystick or an RFID card reader, which publishes a VSS-aligned JSON payload to the InVehicleTopics MQTT broker.
- **Translation Bridge:** A grpc-mqtt-bridge service translates the MQTT payload into gRPC Val/Set updates, updating the Kuksa Databroker running under Eclipse Ankaios.
- **CAN Emission:** The Kuksa CAN Provider maps these VSS broker values to a physical CAN database, emitting a BlinkerCommand .
- **ECU Execution:** The Arduino Uno (MCU1) processes the CAN frame, illuminates the corresponding LEDs, and returns a BlinkerStatus frame over CAN to confirm execution.
- **State Feedback:** The Kuksa CAN Provider captures the status frame from the bus and writes the updated status back to the Databroker, completing the feedback loop.

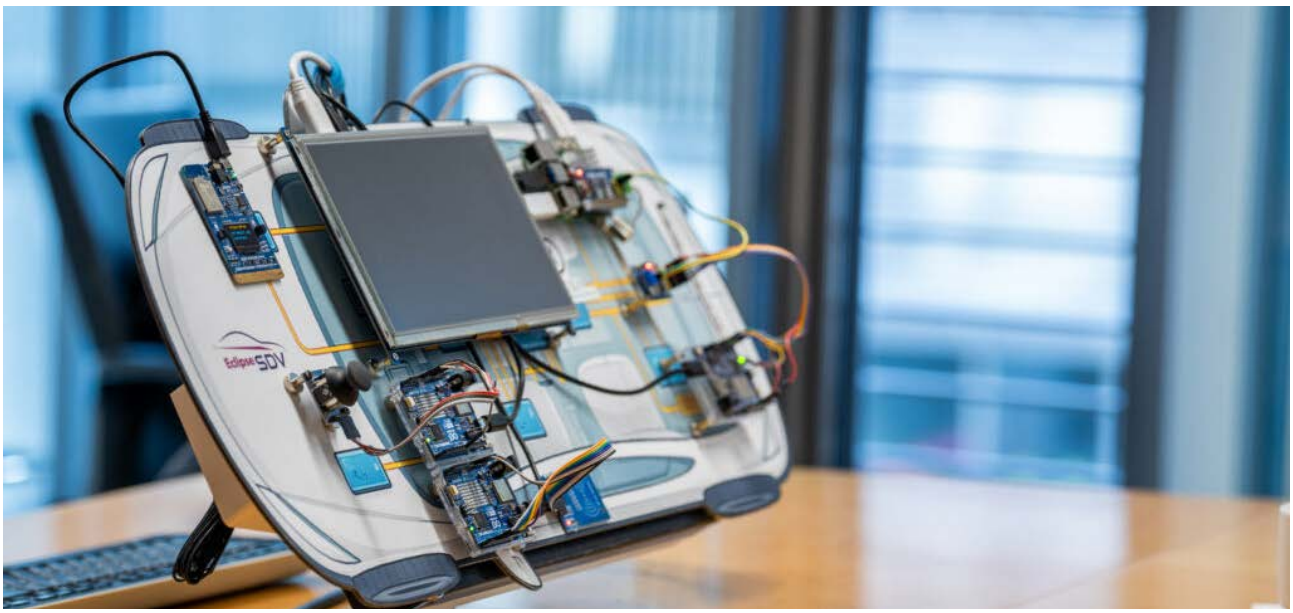


Figure 5: Physical demonstration of the E2E Demonstrator Blueprint

[13] eclipse-sdv-blueprints/e2e-vehicle-signals – GitHub , https://github.com/eclipse-sdv-blueprints/e2e-vehicle-signals README.md – eclipse-sdv-blueprints/e2e-vehicle-signals – GitHub, https://github.com/eclipse-sdv-blueprints/e2e-vehicle-signals/blob/main/README.md

The governance model: lightweight processes and autonomy

A significant challenge in open source consortiums is the administrative overhead associated with project onboarding and lifecycle management. To mitigate this, the Eclipse SDV Blueprints project implements a lightweight governance process that maximises developer autonomy while maintaining strict compliance with Eclipse Foundation standards.

The lifecycle of a blueprint progresses through five structured yet rapid operational stages:

- **The proposal stage:** Proposers announce their project concept by opening a detailed issue in the blueprints repository with system architecture diagrams, and sequence flows to outline the integration.
- **Promotion process review:** The proposal is evaluated against minimal technical criteria, which require running code in a public repository, integration with at least one core Eclipse SDV project, defined committers, and compliance with Eclipse licensing.
- **Comments and peer review:** Community developers and existing committers provide feedback via the GitHub issue, clarifying architectural dependencies and troubleshooting potential integration conflicts.
- **The voting stage:** Existing committers, SDV Steering Committee members, and Eclipse leaders cast their votes directly within the issue.
- **Onboarding and provisioning:** Once approved, the blueprint is provisioned with a dedicated repository under the blueprint organisation and its committers.

To prevent repositories from becoming stale archival sites, the project mandates that proposals are backed by active maintainers.

To incentivise cross-project collaboration, the initiative utilises an automated *SDV Blueprint Badge* system. Participating projects are tracked via metadata and when a project is integrated into a blueprint, the badge is awarded automatically, which directly contributes maturity credits towards the overall SDV maturity initiative.

Code-first design: executable and reproducible

The foundation of the blueprints project rests on a "code-first" philosophy, ensuring that every architectural pattern is immediately runnable. This pragmatic focus shifts the developer experience from studying theoretical documentation to actively deploying and executing containerised stacks. By standardising virtualisation and container runtimes, the initiative lowers the barrier to entry for software vendors.

This execution model relies heavily on technologies such as Docker Compose, Podman, and VSCode DevContainers to construct self-contained environments. For instance, the Fleet Management Blueprint packages its entire multi-tier architecture – comprising telematics forwarders, databases, and telemetry visualisation tools – into Docker Compose files, allowing deployment in a single command. Similarly, the Companion Application blueprint utilises VSCode DevContainers to establish pre-configured development environments with all necessary compiler toolchains, enabling developers to modify, build, and deploy software without manually configuring local runtimes.

Where physical hardware represents a bottleneck, blueprints leverage virtualisation layers to emulate target edge environments. This virtual-first workflow means that a developer can simulate vehicle signal interactions, run continuous integration suites, and validate orchestration logic on generic x86 or ARM cloud nodes before deploying to physical hardware platforms.

Future horizons and community co-innovation

The structured composition of the Eclipse SDV Blueprints initiative reveals critical causal relation-

ships that are actively steering the future of automotive systems engineering. By promoting the use of related efforts such as COVESA VSS as a semantic data standard, the ecosystem allows that vehicle variables interoperate, whether simulated via CSV playback in the Fleet Management blueprint, modified inside a Velocitas-designed application, or triggered by a physical joystick on an Arduino. This helps us create a testbed of innovation that enables developers to decouple application logic from physical ECUs, allowing software components to be reused across different vehicle classes.

This modularity is particularly evident in upcoming, complex industrial use cases.[14] The newly provisioned Commercial Vehicle stack (commercial-sdv-stack) builds directly on the Fleet Management Blueprint, adapting it to support distributed software orchestration via Eclipse Symphony and a universal service mesh via Eclipse uProtocol. This stack is designed to address complex commercial vehicle challenges, such as implementing secure authentication and authorisation across Eclipse Ankaios, Kuksa, and OpenSOVD to meet emissions monitoring and reporting standards.

Similarly, the transition of the "CarMate" project, an award-winning Hackathon prototype integrating ThreadX, uProtocol, Zenoh, and Agentic AI, into an active blueprint proposal demonstrates how hackathons function as rapid innovation pipelines.[15] This pipeline model allows the community to quickly evaluate and adopt emerging edge-AI and real-time operating system architectures.

Ultimately, the collaborative, code-first design of these blueprints ensures that the automotive industry has access to a reliable, vendor-neutral, and pre-integrated suite of software building blocks, minimising redundant engineering efforts and providing a sustainable foundation for the future of software-defined vehicles.

Key takeaways

→ **From concepts to code**: Eclipse SDV Blueprints transform architectural patterns into runnable, reproducible reference implementations.

→ **Integrated by design**: Blueprints combine multiple Eclipse SDV projects to validate interoperability and real-world use cases.

→ **Code-first development**: Containerised, executable examples lower adoption barriers and accelerate experimentation.

→ **Continuous improvement**: Integration issues discovered in blueprints are fed back into upstream projects, strengthening the entire SDV ecosystem.

[14] Blueprint Proposal for SD(C)V – Commercial-Vehicle-driven SDV use cases #21 - GitHub, https://github.com/eclipse-sdv-blueprints/blueprints/issues/21

[15] SDV Hackathon 2025 Outcome – CarMate · Issue #19 · eclipse-sdv-blueprints/blueprints, https://github.com/eclipse-sdv-blueprints/blueprints/issues/19



The Eclipse SDV orchestration landscape

Turning SDV projects into a symphonic platform

BY NACI DAI

As a premier archetype of a distributed system, the SDV commands a codebase exceeding 150 million lines of code that spans from safety-critical microcontrollers to high-performance computing (HPC) clusters and cloud environments. Within the Eclipse SDV Working Group, a sophisticated suite of orchestration technologies has crystallised, featuring projects such as Eclipse Symphony, Ankaios, BlueChi, Muto, Pullpiri, and the nascent Timpani. Although this diversity is often misperceived as architectural fragmentation, it directly mirrors the multifaceted nature of automotive software; no single tool can reconcile real-time determinism with cloud-scale flexibility. Instead, these projects function as specialised, complementary solutions for mixed-criticality workloads and fleet-level management.

Crucially, the Eclipse SDV Blueprints initiative (page 8) bridges the critical execution gap by delivering the runnable, code-centric recipes required to synthesise this diverse ecosystem into an integrated, production-ready reality.

The orchestration stack: a map

In Eclipse SDV, six orchestration projects coexist under one working group. These are vendor-neutral orchestration primitives – parallel experiments exploring genuinely different facets of a problem too large for any single tool. A container orchestrator for an HPC, a deterministic controller for a safety-certified domain, an ROS graph reconciler for an autonomy stack, and a fleet-scale campaign engine in the cloud are not the same product. They differ in language, latency budget, certification posture, failure model, and API surface – because the domains they serve differ in exactly those dimensions.

Rather than proposing yet another reference architecture, the landscape organises into three cooperating planes:

Plane	Role	Projects
Meta-orchestration	Orchestrating the orchestrators across the vehicle-to-cloud continuum: fleet campaigns, staged rollouts, declarative state seeking	Eclipse Symphony
Domain control & state management	Translating high-level vehicle context and OEM policy into concrete activation logic for a specific domain	Eclipse Muto (ROS), Eclipse Pullpiri (vehicle services)
Workload execution & actuation	The in-vehicle actuators that start, stop, schedule, and supervise processes	Eclipse Ankaios (containers on HPCs), BlueChi (systemd services), Timpani (deterministic scheduling)

The challenge is not choosing between these components but integrating them: building the glue – Symphony providers, state managers, telemetry bridges – that links their APIs and lifecycles into a system capable of mixed-criticality deployments at fleet scale.

The projects: strengths and focus

Eclipse Symphony – the maestro

Symphony[16] is a technology-agnostic workflow and lifecycle engine for intelligent edge systems. Its strengths: a **unified object model** (Targets, Solutions, Instances, Campaigns, Catalogs) defining units of orchestration regardless of runtime; **declarative state seeking**, giving resilience across intermittent vehicle connectivity; and a **provider abstraction** translating generic lifecycle APIs into runtime-specific commands – Kubernetes, Docker, Helm, MQTT, or a custom in-vehicle agent. Symphony does not replace in-vehicle orchestrators; it commands them, turning a heterogeneous fleet into a single control plane.

Eclipse Ankaios – containers for automotive HPCs

Ankaios[17] (see page 23) is a workload orchestrator purpose-built for embedded and automotive platforms. Written in Rust, it uses a lean server–agent architecture over Podman or containerd, with declarative manifests, dependency-aware startup, drift correction, and native SDKs. Where Kubernetes optimises for elastic datacenters, Ankaios optimises for **deterministic behaviour and minimal overhead** on vehicle computers — a Kubernetes-shaped idea rebuilt for a car.

[16] https://github.com/eclipse-symphony/symphony

[17] https://github.com/eclipse-ankaios/ankaios

Eclipse BlueChi – determinism for regulated domains

BlueChi[18] is a multi-node **systemd service controller** written in C, aimed at regulated ecosystems requiring functional safety. It extends systemd's D-Bus API across a predefined set of nodes – no dynamic clustering, no scheduler heuristics – and manages containers through Podman Quadlets. Its minimalism is its strength: strict, auditable execution without embedded state logic makes it certifiable where general-purpose orchestrators are not.

Eclipse Muto – declarative orchestration for ROS

Muto[19] brings cloud-native management to ROS 2 stacks on robots and autonomous vehicles. Its Agent maintains cloud connectivity (MQTT, Symphony), its Core synchronises state with an Eclipse Ditto digital twin, and its Composer reconciles the running ROS graph against a declarative stack model – building, validating, and launching nodes, with rollback on failure. Muto's focus is **runtime adaptivity**: dynamic ROS graphs managed as versioned, remotely deployable models.

Eclipse Pullpiri – scenario-driven vehicle services

Pullpiri[20] (Rust) is a vehicle service orchestrator that activates **pre-defined scenarios and policies based on vehicle context** – status, environment, connected devices. It deliberately does not reimplement container runtime operations; it layers scenario logic above them, enabling post-release UX changes through text-defined scenarios rather than code redeployment.

Eclipse Timpani – the drumbeat

Timpani[21], proposed by LG Electronics, addresses what none of the above do: **deterministic, time-triggered scheduling** with end-to-end deadline guarantees for sequential applications spanning ECUs – its motivating example is a rear-parking alert chain of sensor, calculation, and alert apps under hard deadlines. It pairs naturally with Pullpiri: orchestration decides what runs where; Timpani guarantees when it runs.

Laid side by side, the "overlap" dissolves: Eclipse Symphony has no in-vehicle runtime; Eclipse Ankaios and BlueChi occupy different criticality tiers and are deployed together; Eclipse Muto and Pullpiri manage different domains of state; Eclipse Timpani adds a scheduling dimension the others intentionally avoid.

The role of Eclipse SDV Blueprints

Eclipse SDV Blueprints turn this landscape from a parts catalog into an adoption pathway. Eclipse SDV is explicitly a **code-first** community: automotive-grade building blocks, not theoretical frameworks. A blueprint is an **executable reference example**: complete, architected, documented, reproducible code integrating multiple Eclipse projects around a high-value use case. Three properties make them effective:

They are the connective tissue. The Software Orchestration Blueprint[22] integrates Ankaios and BlueChi in a Smart Trailer scenario – trailer connects, a signal fires, the orchestrator dynamically provisions the trailer applications. The ROS Racer Blueprint chains Symphony, Muto, and ROS 2 for OTA updates of driving algorithms, with Ditto digital

[18] https://github.com/eclipse-bluechi/bluechi

[19] https://projects.eclipse.org/projects/automotive.muto

[20] https://github.com/eclipse-pullpiri/pullpiri

[21] https://github.com/eclipse-timpani/timpani

[22] https://github.com/eclipse-sdv-blueprints/software-orchestration

twins and rollback on failure. The glue code these blueprints embed – Symphony providers, reference state managers – is the initial, proven implementation that can later be matured and upstreamed as reusable platform assets.

They are living integration testbeds. Because independently developed projects meet inside a blueprint, integration defects surface there first and fixes propagate back upstream – cross-project quality assurance no working group document can provide.

A more detailed overview of the Blueprints is provided on page 8.

Conclusion

The Eclipse SDV orchestration landscape is not fragmented – it is layered. Symphony conducts; Muto and Pullpiri interpret context into intent; Ankaios, BlueChi, and Timpani actuate with the criticality guarantees their domains demand. What the ecosystem needed was never consolidation into a single tool, but demonstrated composition – and that is what blueprints deliver.

Every project above ships runnable code, and the integrations between them are demonstrated in reproducible environments – Docker/Podman Compose, DevContainers, QEMU – that a developer can execute on a laptop before touching target hardware.

The "Fifty Shades of SDV: A Blueprint-Driven Roadmap for Orchestration" talk[23] at Open Community for Automotive (OCA2026) and the companion repository makes another concrete point, implementing four progressive levels of orchestration:

- **Local vehicle orchestration** – Eclipse Ankaios managing Podman workloads from a declarative manifest.
- **Single-vehicle OTA** – Eclipse Symphony deploying Solutions to an Ankaios target and to a BlueChi/Quadlet vehicle over MQTT: one control plane, two execution engines.
- **Fleet-wide OTA** – Eclipse Symphony Campaigns and Catalogs rolling reusable deployment templates across a fleet with selectors and parallel execution.
- **Canary deployment with automated rollback** – a staged rollout where live telemetry (MQTT, Telegraf, InfluxDB, Grafana) gates progression: healthy canaries advance the campaign fleet-wide; a faulty package triggers automatic restoration of the previous safe configuration via state seeking.

This is the mixed-criticality OTA problem OEMs face in production – containerised workloads updated alongside deterministic services, validated in waves, with first-class rollback – solved with open source components and a few hundred lines of glue.

Industry adoption will not come from another working group or reference architecture. It will come from working, integrated examples that OEMs can clone, run, and extend into products. When the community anchors its collaboration on blueprints, the perceived chaos of fifty shades resolves into a coherent score – and the collection of instruments becomes a symphonic platform, finally turning the promise of the SDV into industrial reality.

Other references:

McKinsey, *Rewiring car electronics and software architecture* (2021)

[23] N. Dai, O. Kral, Fifty Shades of SDV (OCX 2026) https://github.com/nacidai/ocx-fiftyshades

Key takeaways

→ **Layered orchestration**: Each Eclipse SDV project serves a distinct orchestration role.

→ **Complementary tools**: Eclipse Symphony, Ankaios, BlueChi, Muto, Pullpiri, and Timpani work together.

→ **Blueprint-driven integration**: Runnable examples connect projects into complete SDV solutions.

→ **Production-ready OTA**: Open source orchestration enables scalable, reliable software updates.



Eclipse Ankaios

Highly dynamic in-vehicle software orchestration

BY KALOYAN RUSEV

A software-defined vehicle is no longer a product that is flashed once and then left unchanged until the next service visit. It is a distributed computing platform with dynamic feature sets, regional variants, customer-specific packages, and a continuous stream of cloud-driven and signal-driven updates. In this environment, the central challenge is not only how to deploy software, but how to control its lifecycle precisely while the vehicle remains in operation.

That is the problem Eclipse Ankaios[24] is built to solve. Ankaios is a workload orchestrator for modern vehicle compute platforms. It gives OEMs a way to start, stop, update, and reconfigure workloads dynamically, multiple times during an execution phase, without system-wide reboots.

This capability matters when a function must be activated only for a certain time period or region, when a service should come and go based on driving mode, or when a workload must be replaced while the vehicle continues operating.

This dynamic behaviour is the key differentiator. In many embedded stacks, software lifecycle changes are tied to image updates, boot cycles, or vehicle restarts. Ankaios breaks that coupling. It separates workload execution and dynamic reconfiguration from reboot-centric deployment models and enables a vehicle platform to respond to changing conditions with software actions instead of restarts. All of this is done via a single API.

Resource efficiency is another key differentiator for embedded systems. Compared to heavier orchestration stacks, Ankaios requires far less memory and CPU, which directly reduces hardware pressure and leaves more headroom for application workloads.

Use cases include cloud-driven feature activation, signal-driven workload reconfiguration, regional telemetry management, and context-aware AI applications running as Ankaios workloads, like adaptive in-vehicle inference services that are activated only in specific driving situations. In each case, the vehi-

[24] https://eclipse-ankaios.github.io/ankaios/latest/

cle needs a controllable execution layer that can make changes safely and predictably while the rest of the system remains available. Ankaios provides that layer for modern SDVs.

SDV responsibilities, ecosystem integration, and deployment model

Eclipse Ankaios is best understood as the workload lifecycle and execution layer in the Eclipse SDV ecosystem. It is orthogonal to inter-workload communication protocols such as SOME/IP, DDS, CAN, etc. Those protocols define how software components exchange information; Ankaios defines when and where workloads run, how they are started or stopped, and how their desired state evolves over time.

This separation is important, as a vehicle can use the communication stack best suited to a domain or application, while Ankaios remains responsible for lifecycle control.

With its focused responsibilities, Ankaios fits naturally alongside other Eclipse SDV projects in a layered model:

- **Eclipse Kuksa**[25] (page 38) provides signal access, which pairs well with signal-driven workload activation and reconfiguration.
- **Eclipse Symphony**[26] adds fleet-level meta-orchestration, allowing fleet systems to decide which vehicle-level actions Ankaios should execute.
- **Eclipse Automotive Integration for AutoSD**[27] provides an automotive Linux integration baseline on which Ankaios can run, giving OEMs a production-oriented OS environment for managed workloads.
- **Communication middlewares and service interaction layer**:
  - **Eclipse Zenoh**[28] provides efficient pub/sub and query communication across in-vehicle and cloud-connected domains, complementing Ankaios by transporting data between workloads while Ankaios controls their lifecycle.
  - **Eclipse uProtocol**[29] (page 33) provides a common communication abstraction for interoperable in-vehicle services, complementing Ankaios by standardising service interactions while Ankaios governs workload lifecycle.
  - **Eclipse iceoryx**[30]**/iceoryx2**[31] provides zero-copy inter-process communication for high-throughput and low-latency in-vehicle data exchange, complementing Ankaios where communication efficiency is critical.

This combination supports practical SDV integration patterns: an app can consume signals via Eclipse Kuksa, exchange data through communication middlewares such as Zenoh, uProtocol, or iceoryx, and trigger workload lifecycle actions through Ankaios, while Symphony coordinates policy at fleet scale.

The conceptual example in **figure 1** shows how this interaction model can look in practice.

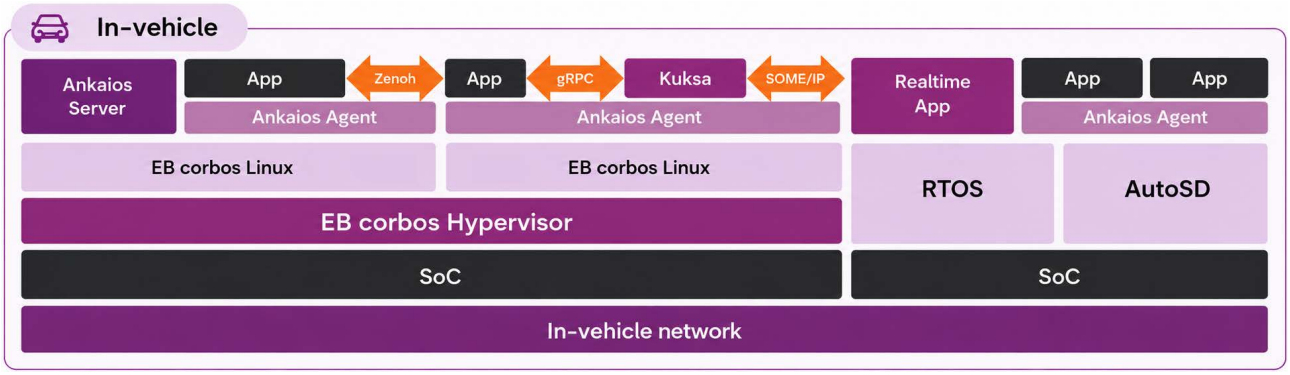


Figure 1: Ankaios example deployment

[25] https://projects.eclipse.org/projects/automotive.kuksa
[26] https://projects.eclipse.org/projects/iot.symphony
[27] https://projects.eclipse.org/projects/automotive.autosd
[28] https://projects.eclipse.org/projects/iot.zenoh
[29] https://projects.eclipse.org/projects/automotive.uprotocol
[30] https://projects.eclipse.org/projects/technology.iceoryx
[31] https://iceoryx.io/

Design principles that matter in vehicles

To cover modern SDV use cases as best as possible, Ankaios is built around a set of design principles that match long-lived, resource-aware vehicle platforms:

- **Declarative desired-state control**: workloads are managed by desired state, which makes behaviour repeatable and auditable.
- **Resource efficiency**: low memory and CPU overhead are essential for embedded vehicle platforms, where compute resources are constrained and hardware cost matters.
- **Clear separation of desired and observed state**: intent and runtime reality remain distinct, enabling transparent drift and failure handling.
- **Deterministic reconciliation**: the same desired state leads to the same orchestration outcome across execution phases.
- **Runtime-agnostic orchestration**: lifecycle management is not bound to a single runtime technology and can evolve with platform needs.
- **Least-privilege by default**: access to lifecycle operations is explicit and policy-scoped per workload.
- **Reboot-independent operation**: workloads can be started, stopped, and reconfigured repeatedly without system-wide restarts.
- **No persistent disk state**: control data is not stored persistently on local disk, reducing flash write pressure, supporting long-term durability, and simplifying recovery through the server-agent model.

Architecture overview

Ankaios follows a clean server-agent architecture (s. **figure 2**). The server is the single source of cluster state, and one or more agents execute workloads on the compute nodes they manage. The server maintains the desired state and the observed runtime state, while agents are responsible for launching workloads through the selected runtime and reporting status back to the server.

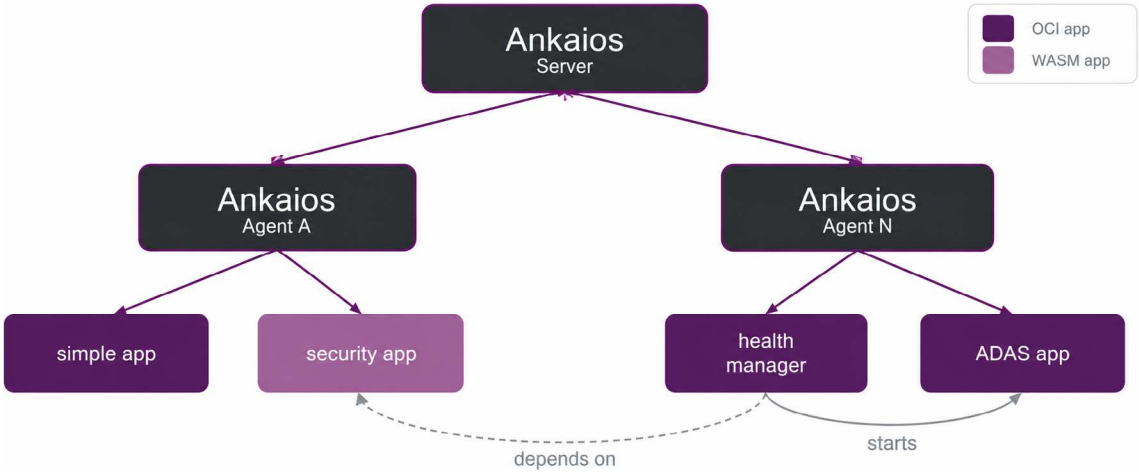


Figure 2: Ankaios cluster high-level architecture

Ankaios is intentionally open in its runtime support. Today, it can execute workloads through Podman and containerd, thereby managing OCI- or WASM-based services. The architecture is designed to remain extensible so that future runtimes can be added without changing the orchestration model itself. This avoids locking the vehicle platform to a single packaging or execution strategy.

Dynamic workload lifecycle control

The main value of Eclipse Ankaios is not merely that it can launch workloads, but that it can control their lifecycle repeatedly and deterministically while the system is live. A workload can be started, stopped, replaced, or reconfigured based on a cloud request, a local signal, a policy decision, or a vehicle state transition. Those changes can happen more than once in the same execution phase, which makes Ankaios useful for scenarios that are too dynamic for reboot-based delivery models.

A common pattern is to keep the base vehicle software stable while individual workloads adapt to context. For example, a telemetry workload may be enabled only in specific markets or in specific situations, a diagnostics collector may be started during service mode, or a feature workload may be swapped out after a backend update.

Because Ankaios is not tied to reboot cycles, it supports operational patterns that are common in software-defined vehicles:

- feature activation and deactivation after a cloud instruction,
- temporary workload startup during diagnostics or service workflows,
- signal-driven activation based on vehicle mode or region,
- graceful workload replacement when a new version is ready,
- coordinated startup ordering for dependent workloads.

Control from within

Beyond external control, the dynamic behaviour described above can also be steered by workloads from within the cluster via the Control Interface. Authorised workloads can read or modify cluster state through Ankaios in a controlled way, allowing a vehicle application to start or stop workloads, inspect state, update desired state, or register for notifications when distinct parts of the complete state change, all without external intervention. This internal control capability makes Ankaios highly adaptable to a wide variety of operational scenarios.

The interface is useful for several general patterns. A cloud connector can translate backend messages into workload actions. A signal-processing workload can trigger a regional telemetry service when a condition is met. A service-mode application can temporarily bring up a diagnostic workload and later remove it again. In each case, the running workload becomes an active participant in the orchestration system, enabling responsive and continuous platform adaptation.

Fine-grained authorisation and transport security

Allowing running workloads to modify cluster state requires a robust authorisation model. Eclipse Ankaios uses a declarative, per-workload access control scheme embedded in the manifest. The default posture is full denial: if a workload configuration declares no *controlInterfaceAccess* rules, the control interface is not mounted into its container at all.

When rules are declared, the managing agent evaluates every Control Interface request before forwarding it to the server. A *StateRule* governs read or write access to portions of the desired state using dot-notation filter masks with wildcard support; a *LogRule* controls access to log streams of other workloads by name pattern. Deny rules take strict precedence over allow rules allowing an easy and intuitive configuration. For example, a fleet connector may hold broad write access while an explicit deny rule protects critical system workloads from modification.

All server-agent and CLI-to-server communication is secured with mutual TLS (mTLS), establishing mutual authentication between server and agents. Each agent's certificate carries its registered name in the Subject Alternative Names field, ensuring that only identified agents can participate in the cluster. The use of ED25519 elliptic-curve cryptography provides strong security properties with smaller key sizes, fitting the resource-aware automotive environment while maintaining high security standards.

Resource efficiency and startup behaviour

Ankaios was designed for the resource envelope typical of automotive-grade compute modules. Benchmarks on equivalent hardware (4 vCPU, 4 GB RAM) show that the platform starts quickly, stays small when idle, and responds efficiently to workload changes (see **table 1**).

These measurements reinforce the architectural point: Ankaios is a lightweight workload orchestrator with a small control-plane footprint and fast lifecycle response.

Metric	Eclipse Ankaios	Lightweight Kubernetes (K3S)
Server startup time	68 ms	480 ms
Idle memory (PSS)	11.2 MB	490.5 MB
Idle CPU	0.2 %	44.9%
Deploy nginx to HTTP	624 ms	2481 ms
Rolling update time	1455 ms	2101 ms

Table 1: Resource efficiency and startup behaviour in Ankaios

Summary

Eclipse Ankaios addresses a central gap in modern SDV vehicles: the need to manage workloads dynamically without tying software change to vehicle reboot cycles. It gives vehicle platforms a workload orchestrator that can start, stop, and reconfigure workloads multiple times during an execution phase, while keeping lifecycle decisions explicit and separating them from inter-workload communication concerns.

Its server-agent architecture, Control Interface, and open runtime model make it a practical execution layer for vehicles that must adapt to cloud instructions, local signals, regional policy changes, or state-based transitions. By integrating cleanly with Eclipse Kuksa for signal access, communication middlewares such as Eclipse Zenoh, uProtocol, and iceoryx for data exchange, and Eclipse Symphony for fleet-level policy coordination, Ankaios fits into the broader Eclipse SDV ecosystem as the vehicle-side workload lifecycle and execution layer.

That is why Ankaios matters: it makes software-defined vehicles operationally flexible without forcing them into reboot-centric workflows, and it does so with a runtime-agnostic, extensible design that is ready for future vehicle execution environments.

Key takeaways

→ **Ankaios enables dynamic workload orchestration** without interrupting vehicle operation.

→ **It provides lightweight orchestration** tailored for embedded systems.

→ **Seamless integration with the Eclipse SDV ecosystem** enables end-to-end SDV solutions.

→ **Declarative, secure orchestration enables scalability** for modern in-vehicle software.



An SDV communication layer with Eclipse Zenoh and Cyclone DDS

Data-centric communication from microcontrollers and vehicle computers to edge and cloud

BY PHANI GANGULA

Imagine the huge amounts of data being exchanged within and across the SDV. Typical examples include camera detections, localisation estimates, and vehicle state and control information. For instance, a roadside unit might send a lane-closure notification. An engineering tool, on the other hand, might need to read specific diagnostic data, while a fleet management service might need to query an event record for analysis purposes. Some data flows would need to remain within the vehicle, while others would need to reach roadside units or edge or cloud infrastructure.

SDV is a system of systems. The data streams move across embedded controllers, domain controllers, high-performance computers, roadside infrastructure, and cloud services. The flows will have to pass through different types of networks: in-vehicle ethernet with bounded latency requirements, shared-memory path locally, constrained embedded buses, intermittent wireless network links, and scalable backend infrastructure. Data has to be delivered to its consumer in the vehicle without exposing all the internal vehicle signals, and failure of any off-vehicle link should not interrupt any internal communication or destabilise a control function of the vehicle.

Eclipse Zenoh[32] and Cyclone DDS[33] complement each other in the realm of SDV communication. Zenoh provides a communication backbone from the vehicle to the V2X, edge and cloud environments. Eclipse Cyclone DDS provides standard-compliant, typed data exchange with the array of Quality of Service (QoS) parameters to build DDS-native systems. The vehicle platform could utilise either technology independently, or couple them at the governed boundary. Zenoh-Pico[34] extends the data-centric communication to resource-constrained devices.

[32] https://projects.eclipse.org/projects/iot.zenoh

[33] https://cyclonedds.io/, https://projects.eclipse.org/projects/iot.cyclonedds

[34] https://github.com/eclipse-zenoh/zenoh-pico

Both projects are open and developed under the Eclipse Foundation umbrella where ZettaScale are the leading contributor to both technologies' development and industrial adoption.

Eclipse Zenoh as a communication backbone for SDV

Eclipse Zenoh offers a unified mechanism for handling data in motion, data at rest, and computations using Pub/Sub/Query abstractions within a shared hierarchical namespace. This allows applications to use the same logical naming scheme to access live and stored data and request results.

A consumer can subscribe to live vehicle state data, query stored values, or request a result from a queryable without embedding the producer's network address in the application. This location transparency decouples the logical data model from the physical one. Data producers can move from an ECU to a high-performance computer, or from the vehicle to an edge device, without requiring every consumer to reconfigure its endpoint address.

The query model offers a capability distinct from continuous publish-subscribe communication. A diagnostic application, for example, can subscribe to live signals while also querying data from an earlier time window through the same key-expression space. A component can also return a computed result only when requested, rather than publishing it continuously. The storage or computation behind the request can run anywhere in the system without changing the logical key expression used by the consumer.

In the roadworks example, an application deployed at the roadside could publish an advisory message to the key `fleet/1/roadworks/a3/km42/advisory`. Vehicles passing through the area could subscribe to that particular key expression without knowing the network address of the roadside application. The selected event could then be published by a vehicle back into the V2X data space, while an edge queryable could provide a computed view of the current state on request. The events could also be stored for future diagnostic or fleet-querying purposes.

Eclipse Zenoh supports peer-to-peer, client-broker, routed and hybrid deployment modes, allowing different communication patterns to be used across the system.

- Applications within an in-vehicle network can communicate directly with one another as peers.
- Devices/applications operating on segmented or resource-constrained networks can connect through a configured Zenoh router instead of depending on multicast discovery or direct peer-to-peer links.
- Zenoh routers can be placed in vehicles, edge locations, and cloud platforms to route selected data according to the system's connectivity requirements and forwarding policies. At network or trust boundaries, these routers can also enforce access-control rules, limiting which participants are allowed to publish, subscribe to, or query specific key expressions.

Zenoh also supports zero-copy shared-memory for intra-host data exchange. This allows multiple applications to access the same payload without repeatedly copying it through the network stack. The greatest performance gains are typically seen with large payloads, particularly when the data is created directly in shared-memory and retained until every consumer has finished using it.

Another notable feature of Eclipse Zenoh is its support for multiple links between Zenoh nodes. When multiple links are established between two nodes, multi-link mode is activated, and an appropriate link is selected based on the message's priority and reliability, as well as the priority range and reliability tags of the available links. These links can operate through different network interfaces and transports, making it possible to distinguish between high-priority operational data and lower-priority bulk data, such as diagnostics or telemetry. This also allows an SDV to use multiple network interfaces, such as Ethernet, Wi-Fi, and cellular connections, simultaneously (**figure 1**).

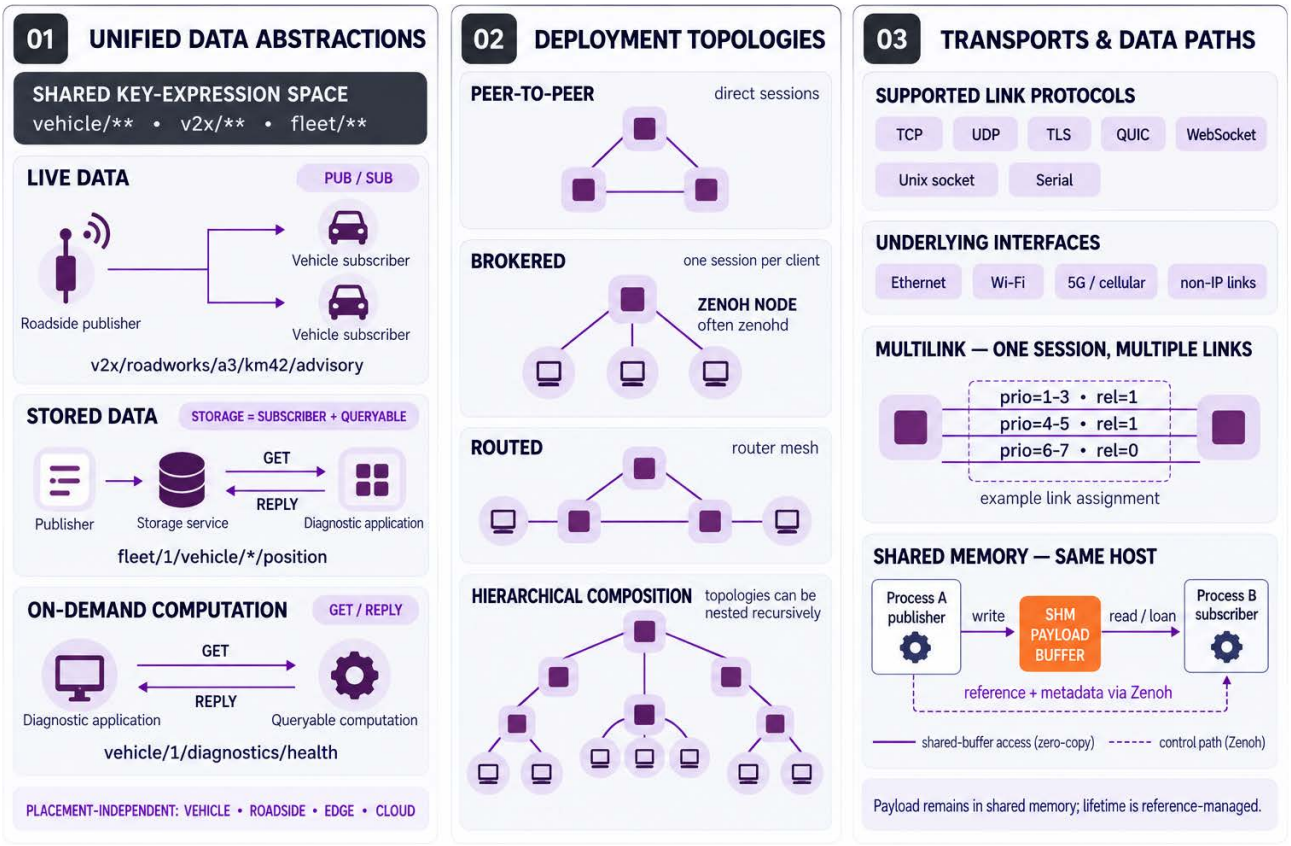


Figure 1: Eclipse Zenoh abstractions, deployment topologies and transport flexibility

The SDV itself is intended to evolve: functions will migrate between ECUs and vehicle computers, services will extend to the edge and cloud, and connectivity options will vary throughout the SDV's lifecycle. This evolution means that the underlying communication technology must be able to accommodate changes within the system without requiring the application layer to adapt to each deployment change. Because the data space is separated from endpoint locations and network topology, the application layer remains stable while the underlying system changes, making Eclipse Zenoh a flexible communication foundation for the ever-evolving SDV.

Zenoh-Pico at the constrained-device edge

Zenoh-Pico is a lightweight version of Zenoh, designed specifically for microcontrollers and other embedded devices with limited resources, and it is implemented in the native C language.

For SDVs, it enables communication between MCU-based sensors, actuators, and controllers with native-Zenoh or Cyclone DDS applications running on ECUs and high-performance-computers. Zenoh-Pico is compatible with bare-metal microcontrollers and supports various embedded environments such as Zephyr, FreeRTOS-Plus-TCP, Arduino, ESP-IDF, and Mbed OS, along with integrations for Raspberry Pi Pico and STM32 platforms that run ThreadX. The available connectivity options depend on the target and configuration, and can include UDP or TCP over IPv4 or IPv6, IPv6 over 6LoWPAN on supported platforms, as well as direct serial links.

Eclipse Cyclone DDS for DDS-native vehicle domains

Eclipse Cyclone DDS is an open-source implementation of the Object Management Group Data Distribution Service standard.[35] DDS applications can autonomously and asynchronously exchange

[35] https://www.omg.org/omg-dds-portal/

data with spatial and temporal decoupling through strongly typed data streams (also known as topics) in the DDS data space (also known as a domain). Discovery and communication between data writers (producers) and readers (consumers) occur when their topic definitions and Quality of Service (QoS) policies are compatible. These QoS policies govern the behaviour of data streams, including their reliability, durability, history, resource limits, deadlines, lifespan, liveliness, and ownership. They make the delivery contract clear for the application and enable individual data streams to express several non-functional properties. A high-rate sensor stream, a periodic vehicle state message, and command message do not have to be treated identically.

Within an SDV architecture, Cyclone DDS provides a data-distribution layer connecting software components running on vehicle computers and across managed in-vehicle networks. Applications for perception, localisation, planning, vehicle-state management and control can exchange strongly typed data directly, without relying on a central broker. Publishers and subscribers are matched through compatible topic definitions and QoS policies rather than fixed endpoint addresses. This allows components to be added, replaced, or moved to a different compute location while keeping their application-level interfaces unchanged (**figure 2**).

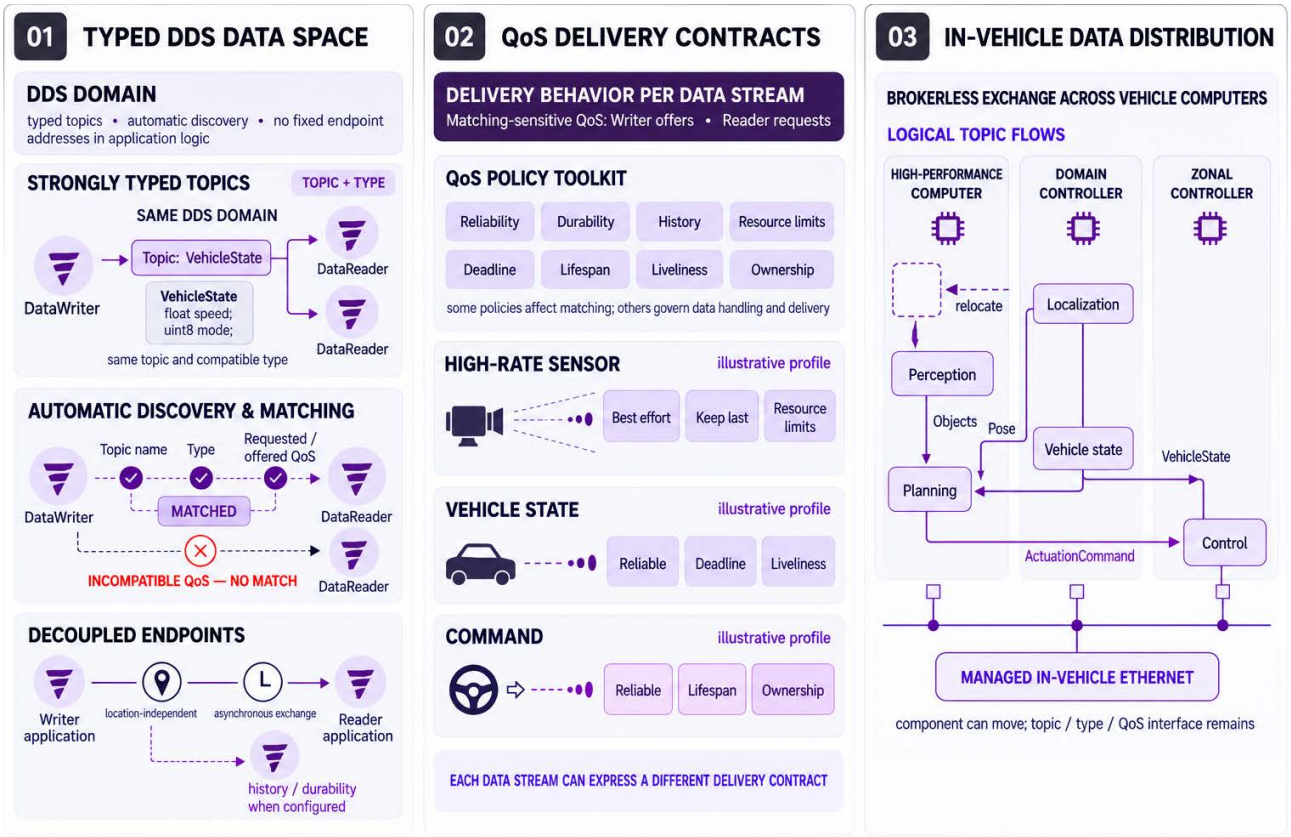


Figure 2: Cyclone DDS typed topics, QoS contracts and brokerless in-vehicle data distribution

Where timing and resource usage are critical, Cyclone DDS provides the mechanisms required to configure and control data delivery. Nonetheless, the middleware is not the sole factor that dictates the resulting end-to-end behaviour. It also relies on the operating system, application scheduling, memory management, network setup, and the hardware platform being used. As a result, end-to-end timing and resource behaviour need to be considered across the complete system level.

Combining Eclipse Cyclone DDS and Zenoh

Eclipse Cyclone DDS and Eclipse Zenoh can be combined in an SDV architecture in which DDS is used for data exchange within local vehicle domains, while Zenoh provides communication across the vehicle, edge, and cloud. Instead of extending DDS discovery and exposing all DDS topics across heterogeneous or intermittent networks, a managed integration boundary ensures that only selected data is shared outside the DDS domain.

The Eclipse Zenoh DDS Bridge acts as an integration boundary. It participates in the DDS domain, discovers readers and writers, routes permitted topics and partitions, and maps them into the Zenoh key-expression space. The bridge forwards serialised DDS samples through Zenoh, allowing them to be routed to another DDS domain or consumed directly by a native Zenoh application, which then decodes the corresponding DDS type definition and serialisation format.

Beyond protocol interoperability, the bridge provides a controlled point at which data can be filtered, rate-limited, secured, or transformed before leaving the DDS domain. This allows existing Eclipse Cyclone DDS-based systems to integrate selectively with broader Zenoh-based architectures while keeping local DDS communication, discovery, and domains isolated by allowing only the required data to pass between multiple DDS domains.

Conclusion

Eclipse Cyclone DDS and Eclipse Zenoh form the basis for communication between various layers of an SDV. Cyclone DDS facilitates the exchange of strongly typed, QoS-controlled information within specified vehicle domains, while Zenoh, together with Zenoh-Pico, extends communication across MCUs, vehicle computers, V2X, edge, and cloud environments. They can be used individually or combined to enable the independent development of SDV layers without redesigning the entire vehicle platform.

Within the Eclipse SDV ecosystem, Zenoh has already been adopted by several projects. Eclipse uProtocol (see page 33) provides transport implementations based on Zenoh, while the Service-to-Signal and Fleet Management Blueprints (see page 8) use Zenoh as a communication backbone. On the DDS side, the ROS Racer Blueprint uses Cyclone DDS, while Eclipse p3com provides an integration path for Cyclone DDS-based systems. Projects developing complete SDV middleware stacks, such as Eclipse S-CORE (see page 53), could consider Cyclone DDS and Zenoh as complementary technologies for their communication layers.

References

- Zenoh protocol: Abstractions – https://zenoh.io/docs/manual/abstractions/
- Zenoh deployment: Deployment modes – https://zenoh.io/docs/getting-started/deployment/
- Automotive: Zenoh automotive and V2X use cases – https://zenoh.io/usecases/
- DDS: Eclipse Cyclone DDS documentation – https://cyclonedds.io/docs/cyclonedds/latest/about_dds/eclipse_cyclone_dds.html
- Integration: Eclipse Zenoh DDS bridge – https://github.com/eclipse-zenoh/zenoh-plugin-dds
- Constrained devices: Eclipse Zenoh-Pico – https://github.com/eclipse-zenoh/zenoh-pico
- Examples: Eclipse Zenoh demonstrations – https://github.com/eclipse-zenoh/zenoh-demos
- Background: Building efficient automotive software frameworks – https://www.automotivetestingtechnologyinternational.com/features/building-efficient-automotive-software-frameworks.html



Why Eclipse uProtocol matters

Generic service mesh for software-defined vehicles

BY PETER LEVASSEUR

The problem: automotive software has been tied too tightly to hardware

Automotive programs were historically organised around hardware. A function was assigned to an ECU, the ECU was chosen for a model, trim, market, or supplier package, and software teams inherited the operating system, bus, middleware, memory budget, and lifecycle. Even within a model year, different trims could mean different compute architectures. The result was branching, porting, repeated integration, and slow feature evolution.

Software-defined vehicles invert that assumption. Value increasingly comes from applications and services, while compute topology keeps changing. A feature may run on central compute in one vehicle and a zonal controller in another, with diagnostics or analytics at the cloud edge. The communication layer must let software remain stable while placement, transport, and topology evolve.

A generic service mesh such as Eclipse uProtocol (see **figure 1**) separates what a service does from how its messages move. uProtocol's stated goal is to connect automotive apps and services "everywhere," routing RPC, publish/subscribe, and notification messages transparently across transports and hosts through URI-based addressing.[36] Its explainer frames the underlying problem as message semantics versus the medium, the "plumbing," that carries them.[37]

[36] Eclipse uProtocol homepage: https://uprotocol.org/

[37] Eclipse uProtocol Explainer: https://uprotocol.org/explainer/

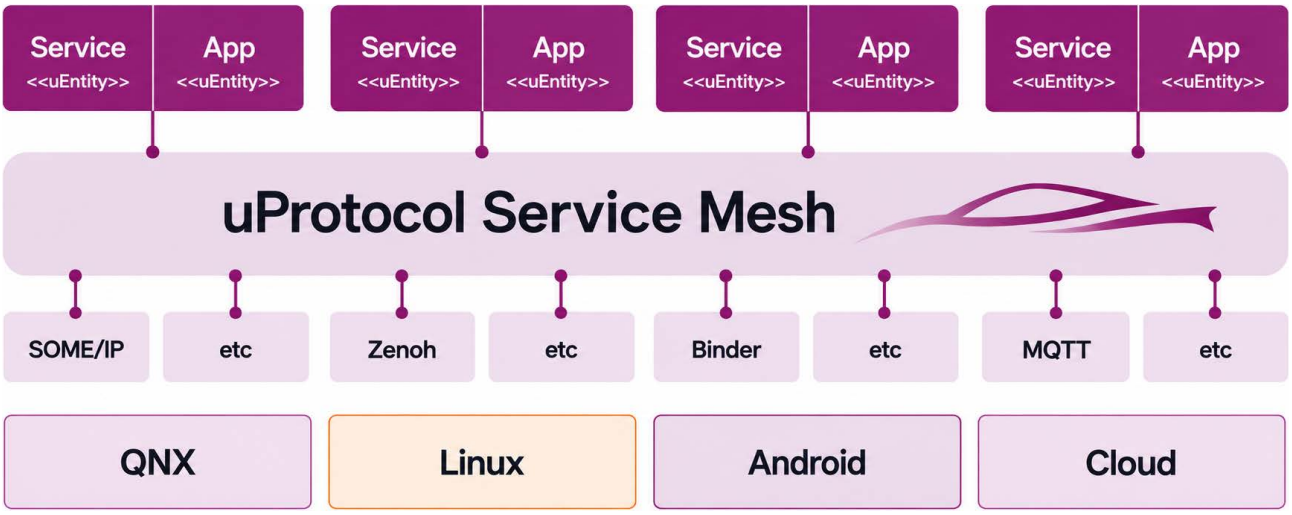


Figure 1: Eclipse uProtocol overview

What "generic service mesh" means in an SDV

In automotive, a service mesh should not mean "one new middleware to replace all old ones." Vehicles span embedded controllers, real-time networks, Linux, Android, and QNX processes, cloud services, mobile apps, and fleet backends. CAN, SOME/IP, Eclipse Zenoh (page 28), MQTT, HTTP, Binder, shared memory, and proprietary stacks all exist for reasons, especially local real-time control. Vehicle platforms, feature allocation and domain-level infrastructure evolve over time, making a one-size-fits-all approach meaningless in all practical terms.

The mesh sits above them, providing a common way to identify services, publish data, invoke operations, subscribe to topics, discover providers, and route messages between domains. Eclipse uProtocol is a transport-agnostic, layered protocol that is deployment, OS, and device agnostic, spanning vehicles, cloud, phones, and charging stations.[38] The explainer is explicit that uProtocol is not another middleware implementation but glue that bridges between the middleware options already in use.[39] Contributed by General Motors in 2023 and developed in the Eclipse SDV Working Group under Apache-2.0, the specification is organised in layers: foundational types, transport, communication, and application-layer platform services.[40]

That layering is what makes the mesh generic. A service can be written against a stable programming model while the underlying transport changes from Eclipse Zenoh to MQTT or Binder to SOME/IP, or as routing moves between vehicle and cloud. "Where the software runs" becomes a deployment and integration decision, not a permanent fork in application code. The spec's guiding principles encode this directly: declare types once and use them everywhere, route messages without inspecting payloads, and decouple applications, services, and platforms so each can be updated independently.[41]

[38] Eclipse uProtocol project page: https://projects.eclipse.org/projects/automotive.uprotocol

[39] Eclipse uProtocol Explainer: https://uprotocol.org/explainer/

[40] uProtocol specification repository: https://github.com/eclipse-uprotocol/up-spec

[41] uProtocol specification, "Guiding Principles": https://github.com/eclipse-uprotocol/up-spec/blob/main/principles.adoc

What uProtocol contributes technically

Eclipse uProtocol gives services a common address. The UUri specification uses URIs to identify devices, software components, topics, and methods. A UUri combines an authority, a uEntity identifier and major version, and a resource identifier; the authority represents a logical deployment location, such as a domain name or a VIN.[42] Applications route to logical services, not to ECU names that change across programs.

uProtocol also defines a common message envelope. UMessage carries metadata in UAttributes plus a typically provided, but optional payload. Attributes include message ID, source, optional sink, time-to-live, type, priority, payload format, and fields for communication status, permissions, tokens, and trace context.[43] Streamers function like IP routers, forward messages on metadata alone and bridge across underlying transports without examining the business payload.[44]

uP-L1 defines the transport API for sending and receiving messages, implemented per transport, runtime, and language. The specification today includes concrete bindings for Eclipse Zenoh, MQTT 5, SOME/IP, CloudEvents, and iceoryx2 shared memory,[45] with language libraries in Rust, C++, Java, Kotlin, and Python.[46] uP-L2 defines the pub/sub, notification, and RPC APIs plus dispatcher roles: uBus within a device, uStreamer between transports, cloud gateway, and device proxy router.[47] uP-L3 adds platform services: uSubscription for subscription management, uDiscovery for locating services, and uTwin for local event caching and digital-twin functionality.[48]

Benefits for passenger-vehicle OEMs

For an OEM, the biggest benefit is flexibility in compute allocation. In a hardware-defined program, teams must predict years in advance which ECU will have the memory, CPU, network capacity, and safety partitioning a function needs. When the guess is wrong, the program pays through redesign, late integration, or branching.

With a generic service mesh, a software module is written against service contracts and communication patterns rather than a fixed transport or host. Across model years or trims, the same service can move as silicon, zonal architecture, or supplier strategy changes while exposing the same interface. Over a vehicle's life, OTA updates change load profiles; if a compute node becomes overloaded, the architecture has a path to relocate, split, or proxy a service, provided timing, safety, and resource constraints are met.

The mesh does not remove engineering discipline: timing budgets, data ownership, safety analysis, cybersecurity review, and integration testing remain mandatory. But employing a service mesh does alter the cost of change: moving a service becomes a controlled deployment problem, not a rewrite of every caller.

[42] uProtocol specification, "uProtocol URI (UUri)": https://github.com/eclipse-uprotocol/up-spec/blob/main/basics/uri.adoc

[43] uProtocol specification, "UMessage" and "UAttributes": https://github.com/eclipse-uprotocol/up-spec/tree/main/basics

[44] uProtocol specification, "Communication Layer (uP-L2)" and "Dispatching": https://github.com/eclipse-uprotocol/up-spec/tree/main/up-l2

[45] uProtocol specification, "Transport & Session Layer (uP-L1)": https://github.com/eclipse-uprotocol/up-spec/tree/main/up-l1

[46] Eclipse uProtocol project page: https://projects.eclipse.org/projects/automotive.uprotocol

[47] uProtocol specification, "Communication Layer (uP-L2)" and "Dispatching": https://github.com/eclipse-uprotocol/up-spec/tree/main/up-l2

[48] uProtocol specification, "Application Layer (uP-L3)": https://github.com/eclipse-uprotocol/up-spec/blob/main/up-l3/README.adoc

Benefits for suppliers and commercial trucking

Commercial trucking suppliers often support low volumes spread across many OEMs, and the integration cost frequently lands on the supplier rather than the OEM. Integrating similar functionality into dozens or hundreds of vehicle programs, each with different gateways, transports, security assumptions, and data models, erodes margin with every bespoke port.

Eclipse uProtocol moves much of that variation to the transport and adapter boundary. The supplier keeps application logic and service contracts stable while swapping uP-L1 transport implementations and deployment configuration per OEM environment. This is not "zero integration": OEM choice of transport, semantics, permissions, lifecycle rules, and safety evidence still matter. But then many OEM projects from the supplier point of view become adapter, conformance, and configuration work instead of one-off rewrites of pieces of software closer to the heart of the business logic.

This is also healthier for the ecosystem. OEMs get more reusable components, suppliers can invest in durable service implementations, and tool vendors can test against common APIs. Key to this is that applications and services carry the customer value, while the plumbing of a service mesh like uProtocol beneath is mission-critical, ubiquitous, non-differentiating, and a natural fit for open source collaboration.[49]

Vehicle-cloud continuity

A useful SDV feature often spans the edge-to-cloud continuum. Depending on whether a cloud provider or an automotive supplier/OEM are talking about it, they tend to emphasise one end of this or another. A mobile app invokes climate control or queries charging status. A fleet operator subscribes to diagnostics. Each path crosses domains with different operating systems, protocols, update cadences, and security controls.

uProtocol targets this flow: invocation and data access work the same from in-vehicle service to in-vehicle service, vehicle to cloud, cloud to vehicle, and app to vehicle.[50] The Eclipse SDV Service-to-Signal Blueprint (see page 12) shows the pattern end to end: a horn service implementing a COVESA uServices interface is exposed over uProtocol via the Eclipse Zenoh transport, interacts with VSS signals through Eclipse Kuksa Databroker, and reaches microcontroller hardware via Zenoh-Pico (see also page 28).[51]

Safety, security, and limits

A service mesh must be used deliberately in a vehicle. uProtocol includes concepts relevant to safe and secure engineering: message priorities, source and sink metadata, permission attributes, transport requirements to preserve metadata, and defined dispatcher error handling with dead-letter reporting.[52] [53] The project strives to support use in safety-critical systems by providing documentation and artifacts relevant to certification,[54] and backs this with machine-checkable rigour: requirements are traced with OpenFastTrace so implementations can demonstrate coverage, and Gherkin scenarios support conformance testing.[55]

These are enablers, not guarantees. A generic mesh is not a safety case, a cybersecurity case, or a hard

[49] Eclipse uProtocol slide deck, "from Service-Mess to Service-Mesh": https://uprotocol.org/uProto-what.pdf

[50] Eclipse uProtocol Explainer: https://uprotocol.org/explainer/

[51] Eclipse SDV Blueprints, Service-to-Signal Blueprint: https://sdv-blueprints.eclipse.dev/docs/service-to-signal/

[52] uProtocol specification, "UMessage" and "UAttributes": https://github.com/eclipse-uprotocol/up-spec/tree/main/basics

[53] uProtocol specification, "Communication Layer (uP-L2)" and "Dispatching": https://github.com/eclipse-uprotocol/up-spec/tree/main/up-l2

[54] Eclipse uProtocol homepage: https://uprotocol.org/

[55] uProtocol specification repository: https://github.com/eclipse-uprotocol/up-spec

real-time scheduler. Local fail-operational loops and deterministic control paths may need specialised mechanisms and should not be forced through a general mesh. The uProtocol explainer says this itself: it is not trying to solve every specialist use case, including real-time critical communication within one local vehicle subdomain.[56]

Security is also governance: the mesh can standardise identity, permissions, and tracing, but credentials, keys, and data-boundary rules must align with the underlying transports and vehicle-cloud operations relevant to the appropriate deployment configuration of the OEM and/or supplier.

Adoption guidance

Start where heterogeneity already hurts: supplier-to-OEM integration, zonal-to-central routing, app-to-vehicle features, signal-to-service translation, and vehicle-to-cloud data flows. A mesh can move messages, but it cannot fix unclear semantics. It works best paired with stable domain models, explicit versioning, contract tests, and shared naming. COVESA VSS and uServices standardise that content layer, and uProtocol reserves uEntity ID space for uServices interfaces.[57] [58]

Conclusion

A generic automotive service mesh lets the industry stop treating every hardware, operating-system, and transport decision as an application rewrite. For OEMs, it supports hardware flexibility, OTA-driven evolution, and reuse across model years and trims. For commercial trucking suppliers, it reduces the cost of adapting stable software to many OEM environments. For the Eclipse SDV ecosystem, it turns non-differentiating plumbing into a shared open source foundation.

Eclipse uProtocol is not a replacement for every in-vehicle protocol or safety-critical control mechanism. Its strategic value is connective tissue: common addressing, message structure, transport abstraction, communication patterns, dispatching, discovery, and subscriptions. In a hardware-defined vehicle, software is fitted to the machine. In an SDV, the machine must be flexible enough to carry the software.

Key takeaways

→ **Hardware-independent software**: Decouples applications from vehicle hardware and transports.

→ **Generic service mesh**: Enables seamless communication across vehicles, cloud, and devices.

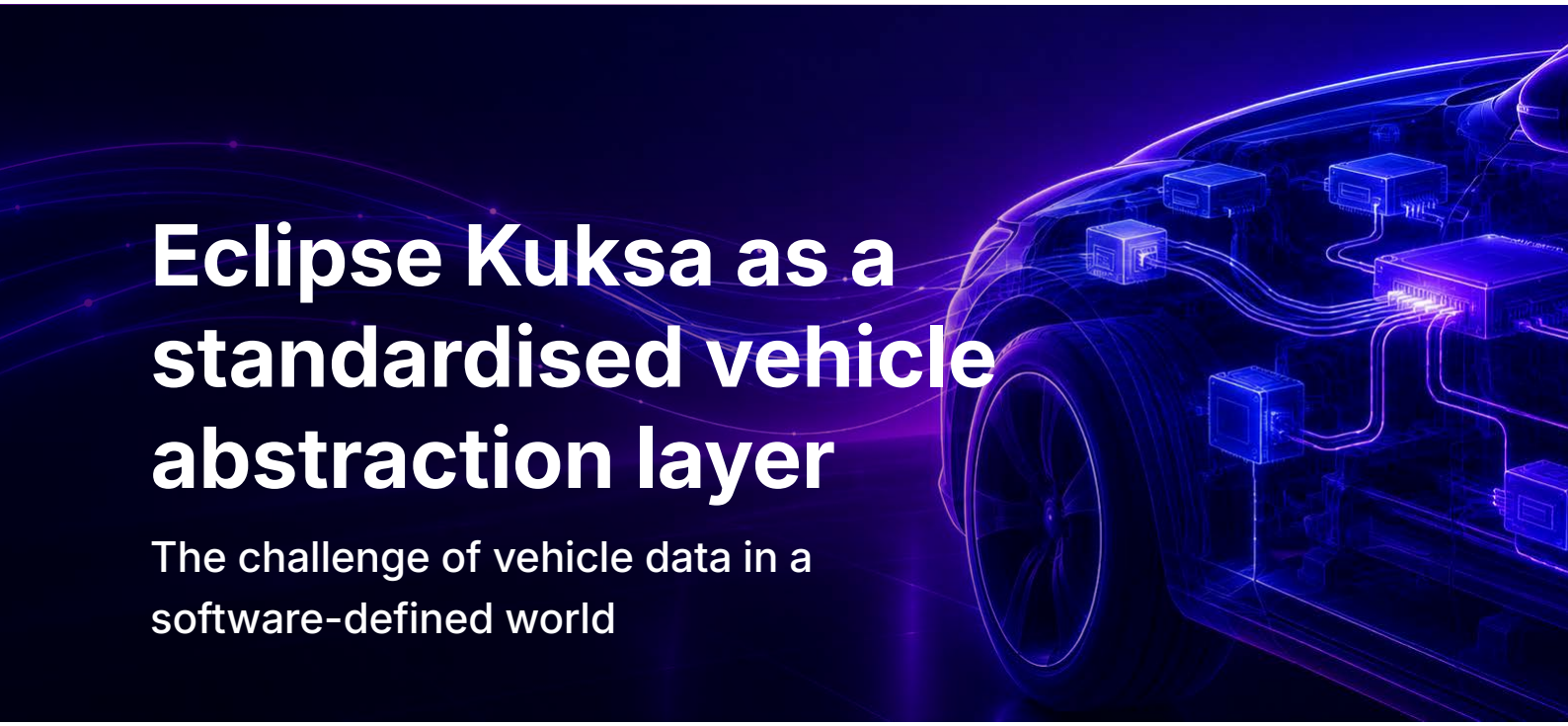
→ **Greater software reuse**: Reduces integration effort for OEMs and suppliers.

→ **Open SDV foundation**: Provides vendor-neutral connectivity for SDVs.

[56] Eclipse uProtocol Explainer: https://uprotocol.org/explainer/

[57] Eclipse uProtocol Explainer: https://uprotocol.org/explainer/

[58] uProtocol specification, “Application Layer (uP-L3)”: https://github.com/eclipse-uprotocol/up-spec/blob/main/up-l3/README.adoc



Eclipse Kuksa as a standardised vehicle abstraction layer

The challenge of vehicle data in a software-defined world

BY SVEN ERIK JEROSCHEWSKI

In an SDV, the ability to access and utilise data from a vehicle's network of sensors and actuators is paramount. However, this data is often locked away in proprietary, low-level systems. Each Electronic Control Unit (ECU) may speak a different "language," using varied protocols and data formats. This lack of standardisation creates a significant barrier for developers, making it difficult to create applications that require "horizontal" access to different sub-systems of the vehicle.

Eclipse Kuksa addresses this fundamental challenge by providing a **Vehicle Abstraction Layer (VAL)** based on vehicle signals. It creates a standardised and secure bridge between the deeply embedded systems of a vehicle and the dynamic, high-level application software. By decoupling software from hardware, Kuksa enables true software portability and accelerates the development of innovative vehicle applications.

What is a vehicle signal?

A modern vehicle is a network of hundreds of sensors and actuators. A **vehicle signal** represents a single piece of data from this network.

Eclipse Kuksa solves the problem of signal fragmentation by adopting the **Vehicle Signal Specification** (VSS), an industry-wide standard developed by the Connected Vehicle Systems Alliance (COVESA). VSS provides an open, industry-standard taxonomy that organises all vehicle data into a single, comprehensive, hierarchical tree structure.

In the VSS, vehicle signals typically fall into three categories:

- **Sensors**: The status of a component, like the current speed (`Vehicle.Speed`).
- **Actuators**: The state of a controllable component, like the tilt of a seat (`Vehicle.Cabin.Seat.Row1.DriverSide.Tilt`).
- **Attributes**: Static or semi-static properties like an identification number (`Vehicle.VehicleIdentification.VIN`).

Instead of dealing with proprietary hex codes or raw bitmasks, developers interact with clean, self-documenting, and standardised paths to ensure that the same information is always represented in the same way. For example, a VSS signal for vehicle speed is defined in a human-readable YAML format like this:

```yaml
# Speed of the vehicle as measured by the wheel sensors.
Vehicle.Speed:
  type: sensor
  datatype: float
  unit: km/h
  description: Vehicle speed.
```

By mandating the use of VSS, Kuksa ensures that developers can write applications against a single, consistent, and well-documented data model.

Decoupling hardware and software for true portability

The core component of Eclipse Kuksa is the **Kuksa Databroker**, a lightweight server written in Rust that runs on a vehicle computer. It acts as the central access point for all vehicle data, exposing the standardised VSS tree to any application that needs it.

This architecture creates a clear separation of concerns:

- **Providers**: These are small, dedicated software components that understand the specific, proprietary language of the vehicle's underlying ECUs and hardware buses (e.g., CAN, SOME/IP). A provider's job is to translate these proprietary signals into the standardised VSS format and "feed" them into the Kuksa Databroker.
- **Applications**: These are the high-level applications (e.g., a navigation app, a custom comfort setting app, or a fleet management tool) that consume vehicle data. Because they communicate with the Kuksa Databroker exclusively using the standardised VSS, they are completely decoupled from the hardware.

This decoupling is the key to **software portability**. An application written to use the `Vehicle.Speed` signal from the Kuksa Databroker will work on any vehicle that has Kuksa installed, no matter how the speed is actually measured or transmitted by the underlying hardware. Developers can "write once, run anywhere," dramatically reducing development time and cost.

Another aspect is that Eclipse Kuksa helps to **break down silos** to enable the creation of more "horizontal applications" that span across multiple subsystems. In traditional architectures, subsystems like the powertrain, or body control, and infotainment are isolated from one another in strict silos. For instance, an infotainment application could not easily read the body status data without custom, complex gateway code. Because the Kuksa Databroker aggregates all VSS-mapped signals under one roof, applications can query and correlate data across different domains (e.g., combining rain-sensor data, suspension status, and vehicle speed) using a single API.

How it works: A concrete example

To make the interaction more concrete, let's consider an example where a driver uses a smartphone app to open the trunk of their car. This interaction flows through the Kuksa Databroker using its gRPC API (`kuksa.val.v2`).

The architecture involves a few key players:

- **Smartphone app**: The user-facing application.
- **In-vehicle app**: An application running on the vehicle's computer, which handles business logic. This could be developed with the Eclipse Velocitas (s. page 42) framework.
- **Kuksa Databroker**: The central data hub.
- **Trunk actuation provider**: A specialised provider that knows how to control the physical trunk latch.

Here is the step-by-step interaction:

1. **User action**: The user presses the "Open Trunk" button on their smartphone app. The app sends a command to the In-Vehicle App.

2. **Application sets a target value**: The In-Vehicle App receives the command and tells the Kuksa Databroker what the *desired* state is. It does this by setting the *target value* of the relevant VSS signal.

python

```
# The app commands the Databroker to set the desired target to True (Open)
client.set_target_value(
"Vehicle.Body.Trunk.Rear.IsOpen", True)
```

---

3. **Provider subscribes and acts**: The `Trunk Actuation Provider` is constantly listening for changes to the *target value* of the trunk signal. When it sees the new target value (True), it executes the hardware-specific commands to unlatch the trunk (e.g., by sending a specific message on the CAN bus).

python

```
TRUNK_IS_OPEN_SIGNAL = "Vehicle.Body.Trunk.Rear.IsOpen"
with kuksa_client.VssClient() as client:
    # Subscribe to changes in the TARGET value
    for updates in client.subscribe_target_values(TRUNK_IS_OPEN_SIGNAL):
        desired_state = updates.get(TRUNK_IS_OPEN_SIGNAL)
        if desired_state is True:
            print("Opening trunk...")
            vehicle_hardware.open_trunk_latch()
                ...
```

---

4. **Provider updates the current value**: After opening the latch, the provider confirms the new state of the trunk (e.g., by reading a sensor on the latch). It then updates the *current value* in the Kuksa Databroker to reflect reality.

python

```
# Confirm the trunk is now open
current_trunk_state = vehicle_hardware.read_trunk_sensor()

# Update the CURRENT value in the databroker
client.set_current_value(TRUNK_IS_OPEN_SIGNAL, current_trunk_state)
```

---

5. **Application gets feedback**: The In-Vehicle App (and any other interested application) can subscribe to the *current value* of the trunk signal. When the value changes to `True`, the app knows the action was successful and can update its own UI accordingly (e.g., by showing a "Trunk is Open" message).

This example demonstrates how Kuksa uses a "digital twin" concept, separating the desired state (target value) from the actual state (current value), to create a robust and flexible system for controlling vehicle functions.

The benefits of a standardised abstraction layer

By embracing a standardised vehicle abstraction layer like Eclipse Kuksa, the automotive industry can realise the full promise of the SDV. The key benefits include:

- **Accelerated development**: Developers can build applications faster, without needing to be experts in the intricacies of vehicle hardware.
- **Enhanced portability**: Applications can be reused across different vehicle platforms, reducing redundant effort and promoting a vibrant ecosystem of third-party app developers. Swapping hardware, sensors, or ECUs only requires updating the low-level provider, leaving the application layer completely untouched.
- **Horizontal integration**: Breaking down proprietary domain silos allows a single application to access data across the entire vehicle seamlessly.
- **A foundation for innovation**: By providing a common ground for data access, Kuksa empowers developers to create and ideate new and exciting features more quickly.

The project has also undergone an external security audit, and the insights gained have been used to further harden the codebase and its APIs.

How to get involved

Eclipse Kuksa is a community-driven project that welcomes contributions from all corners of the automotive and software industries. To learn more or to get involved:

- **Explore the code**: Dive into the Rust-based Databroker and various language clients at the Eclipse Kuksa GitHub Repository.[59]
- **Visit the website**.[60]
- **Inspect real-world blueprints**: See how Eclipse Kuksa integrates with Velocitas (see page 43) or can be used in use cases like fleet management by exploring the Eclipse SDV Blueprints (see page 9).

Key takeaways

→ **Kuksa provides a standardised vehicle abstraction layer** based on the Vehicle Signal Specification (VSS).

→ **It decouples applications from vehicle hardware**, enabling true software portability.

→ **A unified way to interact with vehicle signals** simplifies access across all domains through a unified abstract API.

→ **Kuksa accelerates SDV innovation** by enabling secure, reusable, and and hardware-independent applications.

[59] https://github.com/eclipse-kuksa

[60] https://eclipse-kuksa.github.io/kuksa-website/



How Eclipse Velocitas accelerates SDVs

An end-to-end, modular toolchain for containerised vehicle apps

BY ANDREAS RIEXINGER

The automotive industry is shifting from hardware excellence to the era of the SDV, where vehicle capabilities are continuously enhanced via over-the-air updates. However, traditional embedded development remains held back by proprietary hardware silos and complex legacy architectures. At the heart of the Eclipse SDV ecosystem sits Eclipse Velocitas™,[61] an end-to-end, modular toolchain designed to supercharge the creation of vehicle-independent, containerised vehicle apps.

The background: Why Eclipse Velocitas?

First, it's important to understand the fundamental bottleneck of legacy automotive software engineering.

Historically, writing code for a car required deep knowledge of the target vehicle's underlying hardware and physical topology. If an engineer wanted to create a simple application, such as an automated seat adjustment based on a driver profile, the software had to be tailored precisely to the vehicle's specific wiring, signal matrices, and proprietary communication protocols.

Consequently, porting that application to another vehicle model, or even a refreshed model variant within the same Original Equipment Manufacturer (OEM), required massive code rewriting and validation effort.

The core ambition

Eclipse Velocitas was proposed to establish an open, standardised, and comfortable ecosystem that fundamentally decouples software development from the underlying vehicle hardware. Derived from the Latin word for "speed" or "velocity," Velocitas aims to significantly reduce the time-to-market for automotive features by providing web-era development agility to the embedded automotive world.

[61] https://eclipse.dev/velocitas/. Eclipse Velocitas is a trademark of the Eclipse Foundation. Trademark symbols have been omitted from subsequent references for readability.

By utilising cloud-native patterns, automated pipelines, and standard abstract APIs, Velocitas empowers software engineers to focus entirely on differentiating business logic rather than dealing with complex embedded plumbing. The open source code base hosting these toolchains is fully transparent and maintained by global industry contributors at github.com/eclipse-velocitas.

Architecture and how it works

Eclipse Velocitas acts as a holistic development layer. Rather than being a standalone piece of software running in isolation inside the vehicle, it provides a comprehensive developer experience spanning from the local integrated development environment (IDE) to cloud-based continuous integration (CI) environments and onto the vehicle edge runtime (**figure 1**).

The toolchain and its technical architecture operate through several meshed core components, which can be explored in detail within the official technical reference guide at eclipse.dev/velocitas/docs/about/repository-overview/.

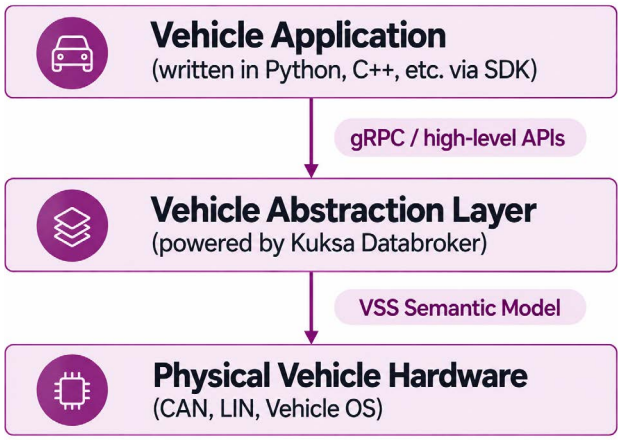


Figure 1: Architectural overview

Vehicle Abstraction Layer (VAL) and Eclipse Kuksa

The cornerstone of the Velocitas philosophy is the complete abstraction of vehicle hardware properties. Velocitas works seamlessly with Eclipse Kuksa, another core Eclipse SDV project (see page 38).

At the center lies the Kuksa Databroker, which organises vehicle data points (sensors, actuators, status values) into a standardised semantic taxonomy known as the Vehicle Signal Specification (VSS), originally initiated by COVESA.

The benefit: Instead of writing complex routines to parse a raw CAN bus frame for vehicle speed, a Velocitas application simply requests the standardised data point `Vehicle.Speed`. The abstract layer translates this request to whatever physical interface the specific car uses.

Language-specific Software Development Kits (SDKs)

Velocitas provides modern SDKs for popular programming languages. These SDKs make accessing vehicle data as intuitive as calling a web API or subscribing to an asynchronous data stream. Developers can write code using modern design patterns, asynchronous loops, and robust error handling natively supported by these languages. The specific language repositories currently include:

- **Python SDK & templates**: The Python programming model[62] can be initialised using the project template.[63]
- **C++ SDK & templates**: For high-performance compiled needs, the native framework components are located on GitHub[64] and its associated template repository can be found there as well.[65]

[62] https://github.com/eclipse-velocitas/vehicle-app-python-sdk

[63] https://github.com/eclipse-velocitas/vehicle-app-python-template

[64] https://github.com/eclipse-velocitas/vehicle-app-cpp-sdk

[65] https://github.com/eclipse-velocitas/vehicle-app-cpp-template

The Velocitas CLI

To tie all these pieces together, the project relies heavily on the Velocitas CLI, which manages the lifecycle, configuration, and dependencies of a vehicle application project. The codebase for this core orchestrating tool is hosted on GitHub,[66] abstracting away setup complexities so developers can focus solely on the business logic.

Containerisation (OCI Compliance)

To achieve complete environmental independence, vehicle apps built with Velocitas are packaged as Open Container Initiative (OCI)-compliant containers (such as Docker or Podman containers). Containerisation ensures that the application carries all its necessary dependencies and can run predictably on any vehicle operating system or middleware that supports container runtimes, such as Eclipse Kanto or Eclipse Leda (see runtime integration specifications[67]).

The pre-configured developer workspace

Setting up an automotive development environment usually takes days or weeks. Velocitas solves this by using Microsoft Visual Studio Code Dev Containers. When a developer clones a Velocitas project template, the IDE automatically sets up a localised virtual Linux environment containing all required toolchains, compilers, linter configurations, and vehicle simulators using optimised base images from GitHub.[68]

The lifecycle: From ideation to vehicle core

What makes Velocitas uniquely powerful is its out-of-the-box continuous integration and continuous deployment (CI/CD) blueprint. It realises a true "code to car" workflow through specific, sequential phases:

Phase 1: Prototyping

Developers can start their journey in rapid prototyping environments like digital.auto, designing feature ideas in a browser-based Python playground using the VSS syntax. Once the basic logic is proven, the prototype can be migrated effortlessly into a local Eclipse Velocitas project workspace. The toolchain utilises the Vehicle Model Generator tool[69] to generate language-specific APIs directly from underlying VSS data definitions.

Phase 2: Local simulation and testing

Inside the VS Code environment, developers don't need a physical car dashboard or heavy hardware-in-the-loop (HIL) simulators to test code. Velocitas spawns mock data providers and local MQTT or gRPC brokers. Developers can simulate changing vehicle parameters (e.g., accelerating speed, opening doors) and immediately observe how their vehicle app reacts and triggers actuators.

Phase 3: Automated CI/CD workflows

When code is pushed to a repository (such as GitHub), Velocitas-provided workflow templates automatically trigger. The pipelines perform a multitude of tasks:

1. Multi-architecture compilation (building container images optimised for both standard x86_64 cloud architectures and energy-efficient in-vehicle ARM64 chips).
2. Automated execution of unit and integration tests against virtual mock layers.
3. Vulnerability and security scanning of the container image using open source utilities like the license auditor.[70]

[66] https://github.com/eclipse-velocitas/cli

[67] https://eclipse-leda.github.io/leda/docs/app-deployment/velocitas/

[68] https://github.com/eclipse-velocitas/devcontainer-base-images

[69] https://github.com/eclipse-velocitas/vehicle-model-generator

[70] https://github.com/eclipse-velocitas/license-check

4. Automated documentation processing, which renders project schemas utilising specialised workflows such as the release documentation action.[71]
5. Automatic publishing of verified release artifacts to a secure container registry.

Best use cases: Where Velocitas shines

Eclipse Velocitas is highly versatile, but its structural focus on isolation, agility, and abstract modelling makes it exceptionally valuable for specific types of enterprise and consumer applications.

Use case category	Practical scenario	Velocitas advantage
Fleet management & telematics	Real-time tracking of commercial vehicle delivery routes, payload conditions, energy consumption analytics, and predictive component wear.	Standardised data access allows logistics enterprises to write a single fleet-tracking app that deploys uniformly across diverse multi-brand commercial fleets.
Smart charging & battery management	Dynamic algorithms controlling charging rates, cabin pre-conditioning based on grid prices, and battery health reporting.	Apps can securely query battery states via VSS and interact safely with cloud charging APIs without interfering with core driving physics.
Comfort & personalisation (IVI Overlays)	"Dog Mode" (maintaining cabin climate control while parked with alerts on user phone), seat and mirror presets mapped to user profiles, or adaptive ambient lighting linked to driving modes.	Rapid iteration cycles allow OEMs to deploy lifestyle and comfort features instantly over-the-air, matching consumer tech expectations.
Edge computing & edge-to-cloud analytics	On-vehicle filtering of high-frequency sensor streams (e.g., cameras, LIDAR, or chassis vibration data) before uploading refined telemetry data to the cloud.	Containerised execution allows intensive data analytics apps to be scaled or throttled without risking safety-critical system resources.

[71] https://github.com/eclipse-velocitas/release-documentation-action

Critical distinction: Where *not* to use it

While Velocitas accelerates modern feature delivery, it is crucial to recognise its architectural boundaries. It is explicitly *not* designed for safety-critical, hard real-time systems.

Crucial architecture boundary:

Core automotive functions – such as powertrain control, electronic braking systems (EBS), airbag deployment, or low-level ADAS sensor fusion – require deterministic execution with microsecond latency and strictly certified functional safety (e.g., ISO 26262 ASIL-D).

Because Velocitas targets containerised execution and abstracts data via brokers, it introduces mild latency overheads that are completely acceptable for comfort, infotainment, telemetry, and smart algorithms, but unsuitable for hard real-time execution loops. It works alongside classic AUTOSAR platforms, not as a replacement for them.

Driving into an open ecosystem

Eclipse Velocitas is a catalyst for a more open and collaborative approach to automotive engineering. By dismantling the traditional, rigid hardware dependencies and bringing modern, cloud-native DevOps principles to the car, it accelerates developer velocity.

For automotive OEMs and Tier-1 suppliers, adopting an open source development toolchain like Velocitas means eliminating redundant engineering costs spent on creating custom, in-house abstraction frameworks. For software engineers from outside the traditional automotive area, it lowers the barrier to entry, transforming the modern vehicle into an open, accessible, and exciting software canvas. As software increasingly defines the driving experience, tools like Eclipse Velocitas ensure that the speed of innovation can finally keep pace with the imagination of the developer.

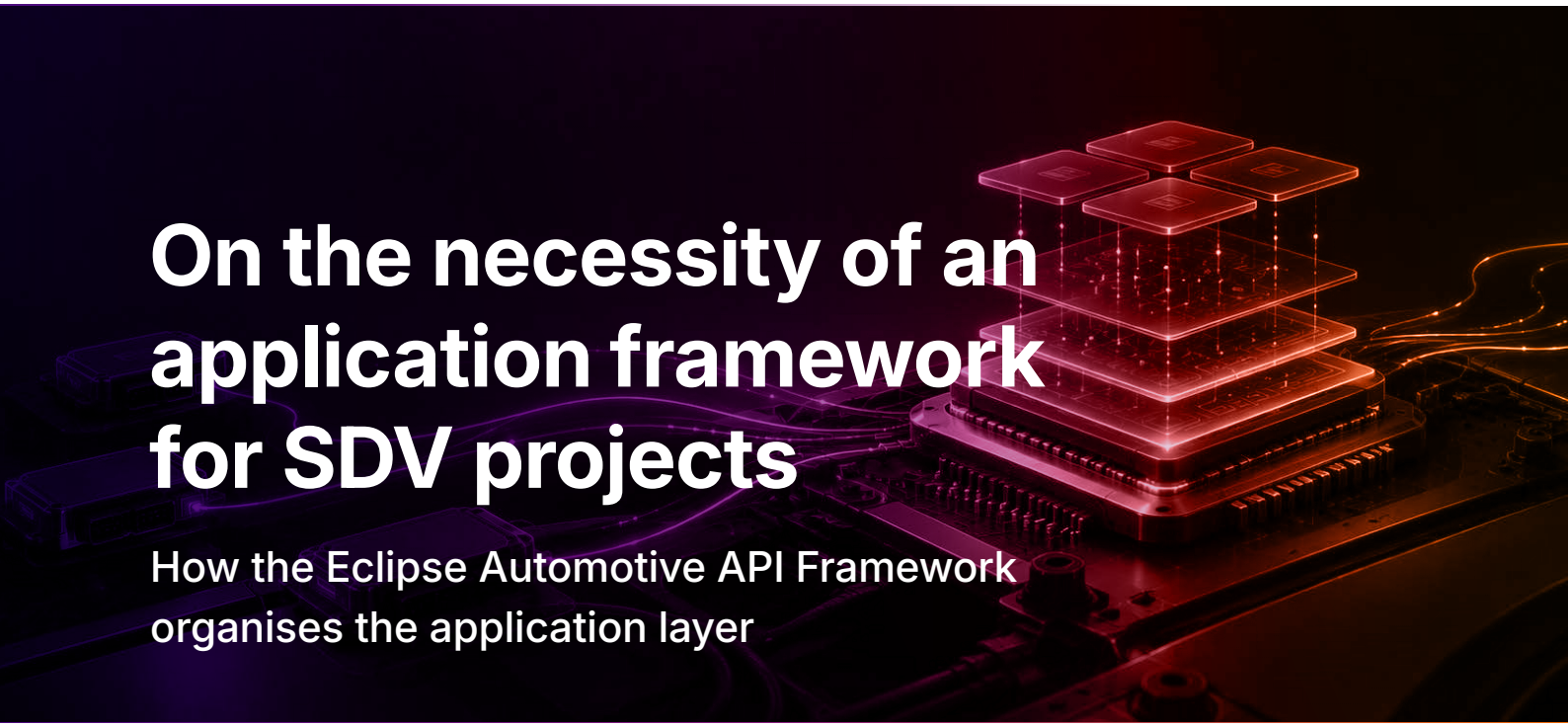
Key takeaways

→ **Eclipse Velocitas accelerates SDV development** with an end-to-end, cloud-native toolchain.

→ **It decouples applications from vehicle hardware** through standardised APIs and Eclipse Kuksa.

→ **Containerised apps enable rapid development, testing, and deployment** across vehicle platforms.

→ **Velocitas brings modern DevOps practices** to automotive software engineering.



On the necessity of an application framework for SDV projects

How the Eclipse Automotive API Framework organises the application layer

BY MANUEL FESSLER

At present, the automotive industry is facing a shift towards value creation that is driven by innovation in software. Accordingly, many players in the sector are in the middle of a transformation to become a software-driven company. In other words, they increase labour and development effort in this area to stay competitive. At the same time, the prevailing vehicle E/E architecture gets increasingly centralised, with one or even multiple High-Performance Computing (HPC) control units at its center (**figure 1**).

An HPC system in this context is usually realised by a System-on-Chip (SoC) device, which unites microcontroller and microprocessor units in one housing. Both have access to shared memory to ensure efficient data exchange between the cores. The microcontroller can act as a safety island or deals with highly deterministic applications through the Real-Time Operating System (RTOS) it operates. Meanwhile, the microprocessor offers the required computational power for applications from autonomous driving or infotainment domain, for example.

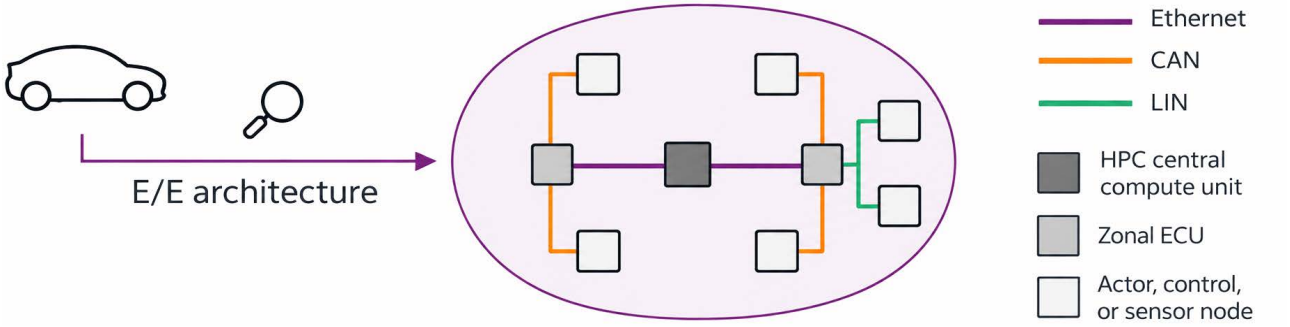


Figure 1: Schematic illustration of an SDV with centralised E/E architecture

Problem statement

Both trends, increasing value creation through software and the shift towards centralised vehicle architecture, go hand in hand and form the guard rails for the often-propagated vision of an SDV. Missing or at least incomplete standardisation in the field of HPC ECUs, however, leads to an immense variety of proprietary solutions at OEMs and suppliers. On top, there are new contributions from the IT domain, the Internet of Things (IoT) world, or the Free and Open Source Software (FOSS) community. The big problem is that all parts that get modeled and developed in different environments need to form one system down the road. As a result, everything must be integrated into ECU-local stacks and the vehicle-wide distributed system eventually. Reality shows, this is a cumbersome endeavour, full of resolving incompatibility and performance issues.

At this point, it is worthwhile having a short look at how software development and integration is accomplished in the world of microcontroller-based ECUs. Two things are important in this regard and visible from **figure 2**.

First, the basic software stack, i.e., everything between the network endpoint and the Software Component (SWC) on the application layer is standardised in AUTOSAR Classic.[72] This means, all building blocks as well as their relation and interfaces are clearly specified.

Second, the Application Programming Interface (API) is kept generic and provided by a completely generated abstraction layer, the Run-Time Environment (RTE). This way, application development can be decoupled from the basic software stack and its integration on the actual hardware platform. Collaboration and work split between different teams and organisations becomes feasible.

Figure 3 provides a schematic illustration of the microprocessor case in comparison, with all software building blocks involved. Here, the situation is different and for the following reasons more difficult to manage.

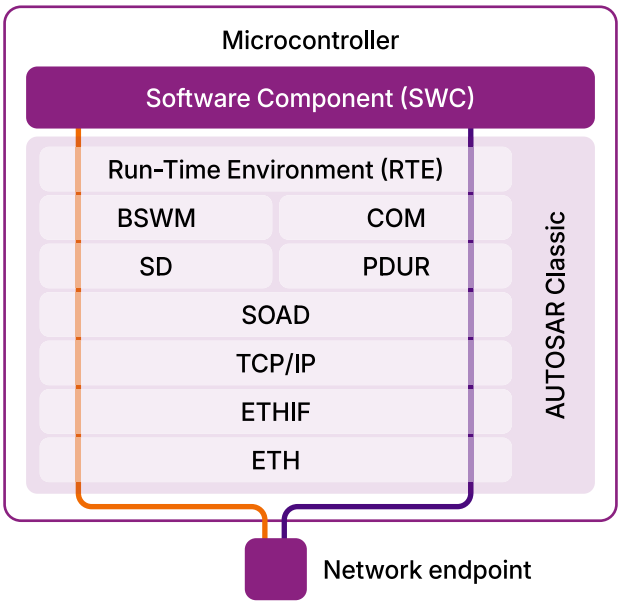


Figure 2: Building blocks of a microcontroller software stack for service-oriented communication according to AUTOSAR Classic

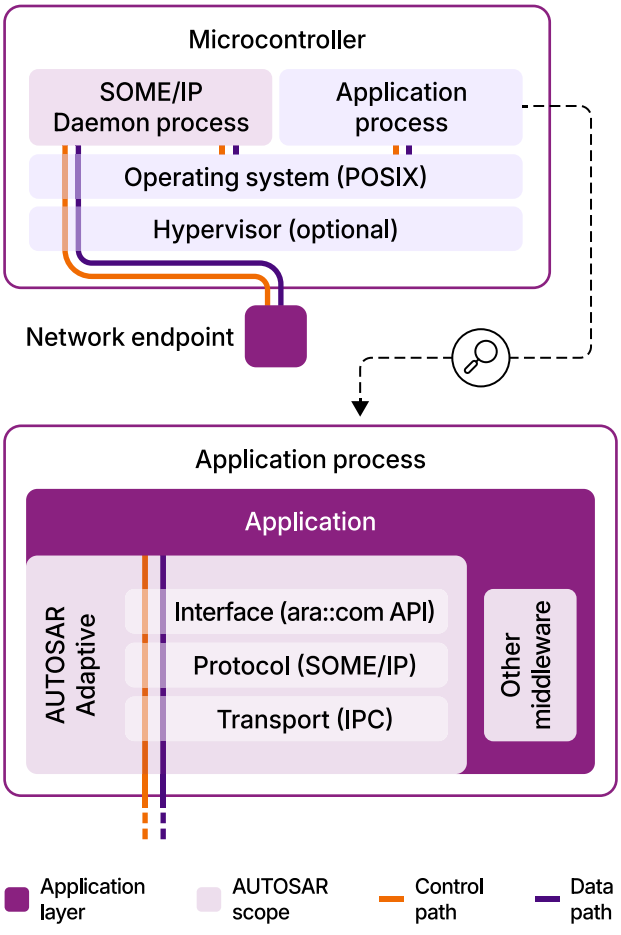


Figure 3: Building blocks for service-oriented communication on a microprocessor according to AUTOSAR Adaptive.

[72] https://www.autosar.org/standards/classic-platform

The first aspect is the POSIX-compliant operating system (OS). It introduces abstraction to hide hardware details from the application layer. In comparison to a monolithic software stack on a microcontroller with one, global address space, a POSIX-based system on a microprocessor is realised in a distributed, flat form. It consists of processes that are all managed by the operating system, each one with its own virtual address space. In consequence, processes must interact with the OS when it comes to lifecycle management or the allocation of resources. For that reason, corresponding integration code, often referred to as boilerplate, is needed and depends on the choice that is made for this layer. With Linux, QNX OS,[73] and other specialised solutions, different options exist. Some are well-suited for the development and evaluation phase. Other alternatives again are better for safety-critical or real-time setups.

Second, several middleware solutions can coexist in one system, all with different API in terms of syntax and semantics. Middleware according to AUTOSAR Adaptive[74] takes care of service-oriented communication via SOME/IP or manages vehicle diagnostics. It further takes safety and security into account. In other words, it provides services for the application layer to interact with the surrounding automotive environment and to communicate with other ECUs in the vehicle. Besides the standardised solution in AUTOSAR, however, there are plenty of middleware alternatives from the IoT and IT world that find their way into development projects. On top and from the automotive open source community, Eclipse S-CORE[75] (see page 53) can be named as an emerging candidate for this layer.

Considering the above-listed aspects leads to the following conclusion: The variety of OS and middleware-related building blocks results in many viable solutions and combinations thereof, which is curse and blessing at the same time. Resulting systems are tailored to specific scenarios, but hardly compatible in most cases. Every choice involves pros and cons for application development. Friction in the integration phase, however, is inevitable as a standardised one-for-all approach as given with AUTOSAR Classic for the microcontroller domain is not available for the microprocessor today.

Altogether, this situation is not satisfying as it means unclear boundary conditions for all parties that work on or interface with the microprocessor application layer. OEMs face a lock-in with technical solutions and must deal with initial creation and maintenance of frameworks and boilerplate for their specific projects. Suppliers, on the other hand, simply do not know which interfaces to rely on for their applications and software products. Collaboration and work share become difficult or even impossible.

Application Framework

One conceivable solution to overcome the situation described is the introduction of an ecosystem for developers that work on the microprocessor application layer. Ideally, such a framework is either standardised or at least widely accepted, for example through open source availability and a related community. The latter way is chosen by the Eclipse Automotive API Framework[76] project. It addresses the above-mentioned challenges by introducing an open source Application Framework[77] with the following characteristics:

The central pattern in the Application Framework is abstraction of the application layer from its lower-layer building blocks, i.e., middleware and operating system (**figure 4**). As abstraction is not for free, this layer is designed to be thin. In other words, minimum code with minimum overhead. To foster

[73] https://qnx.software/en/software/products-and-solutions/qnx-os-and-os-for-safety

[74] https://www.autosar.org/standards/adaptive-platform

[75] https://github.com/eclipse-score

[76] https://projects.eclipse.org/projects/automotive.autoapiframework

[77] https://github.com/eclipse-autoapiframework/application-framework/tree/main/VAF

reuse, the proposed software architecture on executable-level is completely modular. Code artifacts that are provided by the framework go hand in hand with code that is added or integrated by a developer. Application modules are standalone library projects that hold the application business logic and allow development and unit tests out of context. Collaboration and co-working in application development becomes possible by design. Middleware abstraction modules deal with the API and specifics of the lower layer and represent the sole touching point for application modules. Executable-level housekeeping, interaction with the operating system, execution of periodic tasks, and more is covered by a control module, which is available once per executable.

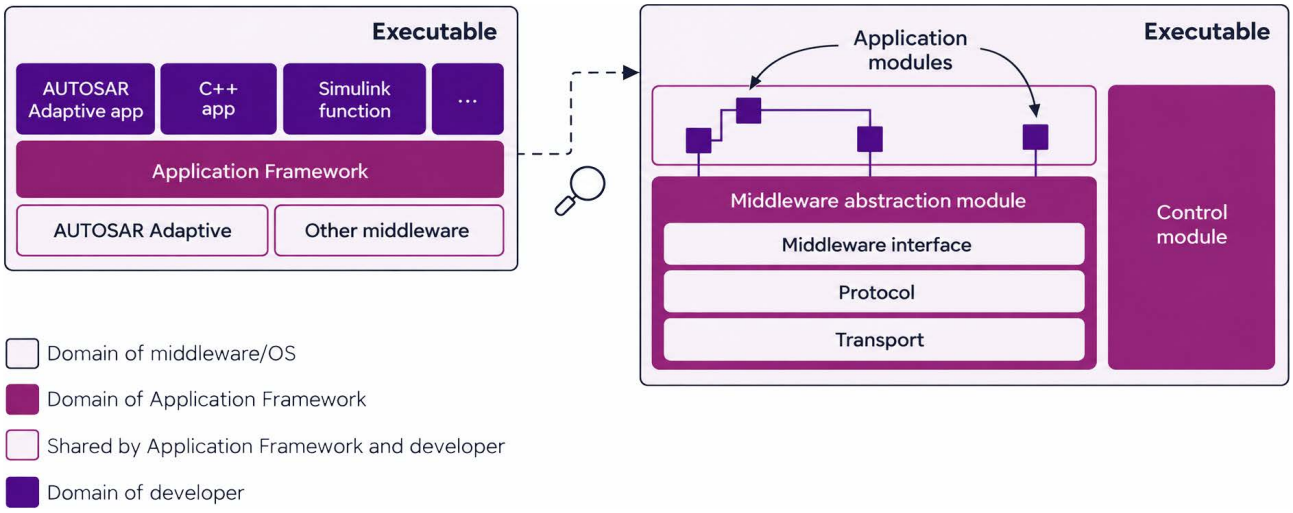


Figure 4 : The Application Framework executable-level architecture

The whole ecosystem follows an unambiguous definition of the term application as follows: An application is of distributed kind and considered a thematic bracket. A lane-keeping application for example. On a technical level it can consist of 1 to N executables. Each executable in turn can consist of 1 to M application modules.

All artifacts and building blocks in the framework are text-based (**figure 5**). The foundation is a data model that captures all relevant attributes of an application. The configuration of this model through a software developer takes place according to the requirements of the project at hand. In practice, it is done via an abstracted user front end in Python (configuration as code). Importers open the solution for reuse and integration of legacy files and formats. Finally, the consolidated data set from user configuration and imported files gets stored in a data exchange format (JSON). From there it is consumed by the generators of the framework, which produce interface definition, middleware abstraction and control module code, implementation and test stubs for the developer, further project artifacts related to build environment, and more. The final puzzle piece is a holistic workflow and associated command-line tool that covers all steps, from project creation to modeling and configuration in combination with import of artifacts, generation, build, and test.

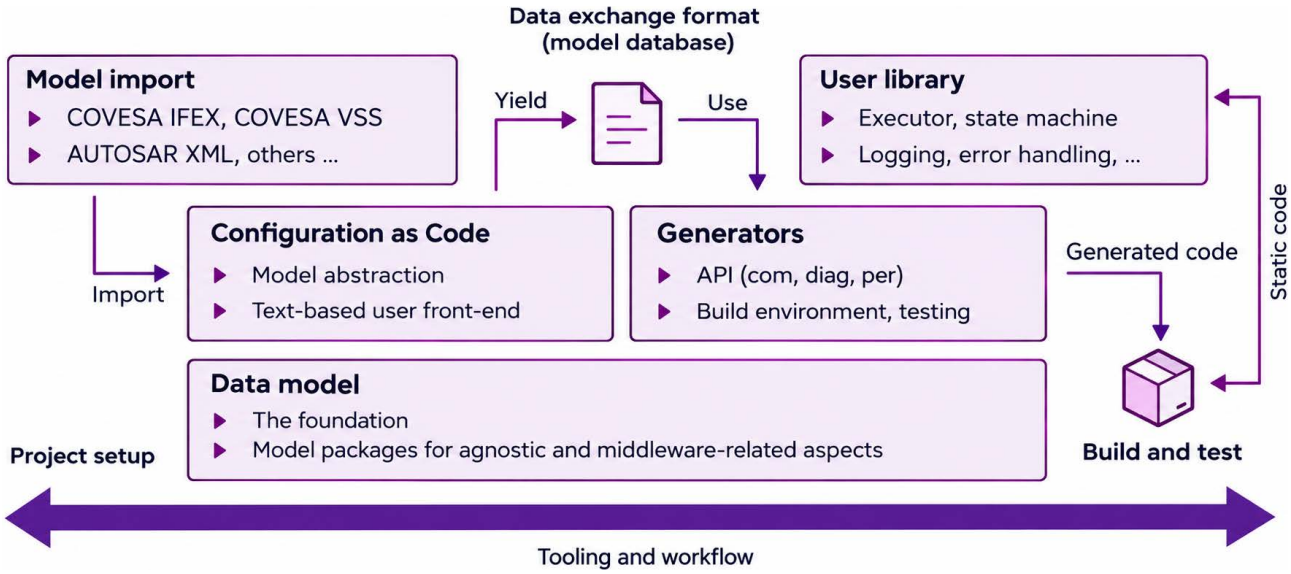


Figure 5: Overview and relation of the Application Framework building blocks

Applied to practice, the Application Framework aims to support different use cases with seamless transition from rapid prototyping to the realisation of a production system.

The initial focus is on prototyping projects, where code-first is a central paradigm. That is, algorithms and business logic are in focus and get developed irrespective of standards and independent of commercial tools. This modern approach comes naturally to engineers and computer scientists without an automotive background. Programming skills and a development environment with helpful plugins are all this audience needs to be productive. AI and agentic support fit naturally into this approach, too. For that reason, the Application Framework avoids automotive-specific concepts and is instead built on generally applicable terms and patterns from computer science. Project templates and a Visual Studio Code extension round off the solution. Setting up a new project and the creation of an executable binary becomes a matter of minutes.

The second area of focus are production projects, which are in complete contrast to the prototyping approach and typically based on some system specification. AUTOSAR Adaptive standard or model-driven development according to MATLAB Simulink[78] can be named as examples in this context. Common to them is the considerable front-loading through the description of the system and its architecture in the form of a model. This abstract description is consumed by code generators that provide artifacts for the application projects that are in line with the model. This approach around centralised system specification, however, contrasts with agile and collaborative work modes as well as short feedback loops.Its monolithic design further opposes the idea of software as a product and can be considered problematic in terms of lock-in to a solution or vendor. The Application Framework addresses this aspect through openness for existing standards and tools with their respective file formats for data exchange.

Conclusion

The field of application development for HPCs in the automotive sector is extremely dynamic. For microprocessor-based systems, in particular, different technical solutions with incompatible interfaces, file formats, patterns, and workflows, exist and compete for projects. Inconsistent terminology further leads to troublesome and fruitless discussions. This setting makes development and integration of software

[78] https://www.mathworks.com/products/simulink.html

building blocks a complicated task. Realisation of series projects remains a hard task.

This situation led to the idea of the Application Framework as part of the Eclipse Automotive API Framework project. The following list of high-level requirements summarises the central aspects it addresses:

1. Application development should be decoupled from the lower-layer building blocks, making them interchangeable. In any case, choosing between them shouldn't be a decision that developers at the application layer need to make.
2. The API should be unified and independent of automotive-specific terminology, allowing developers from any domain to use it without requiring domain knowledge.
3. Application design should be facilitated through appropriate but lean modelling.
4. Integration glue code should be generated for possible middleware and OS combinations to the largest possible degree.
5. Project setup, configuration of build environment, and writing of test stubs are repetitive tasks that should be automated where possible.
6. Artifacts and work-mode should be text-based and therefore compatible with modern agile, code-driven, Git-based, and agentic workflows.
7. Lock-in should be avoided through open source availability. A related community, likewise, driven by OEMs and suppliers, should create a win-win situation and lead to adoption of the ecosystem in the field.

This list underlines the necessity of an application framework in the SDV landscape. The open source solution in Eclipse SDV addresses those requirements. In doing so, it fills this gap and proposes a modern, flexible, and unifying solution that supports design, implementation, and test of distributed applications for microprocessor-based embedded systems.

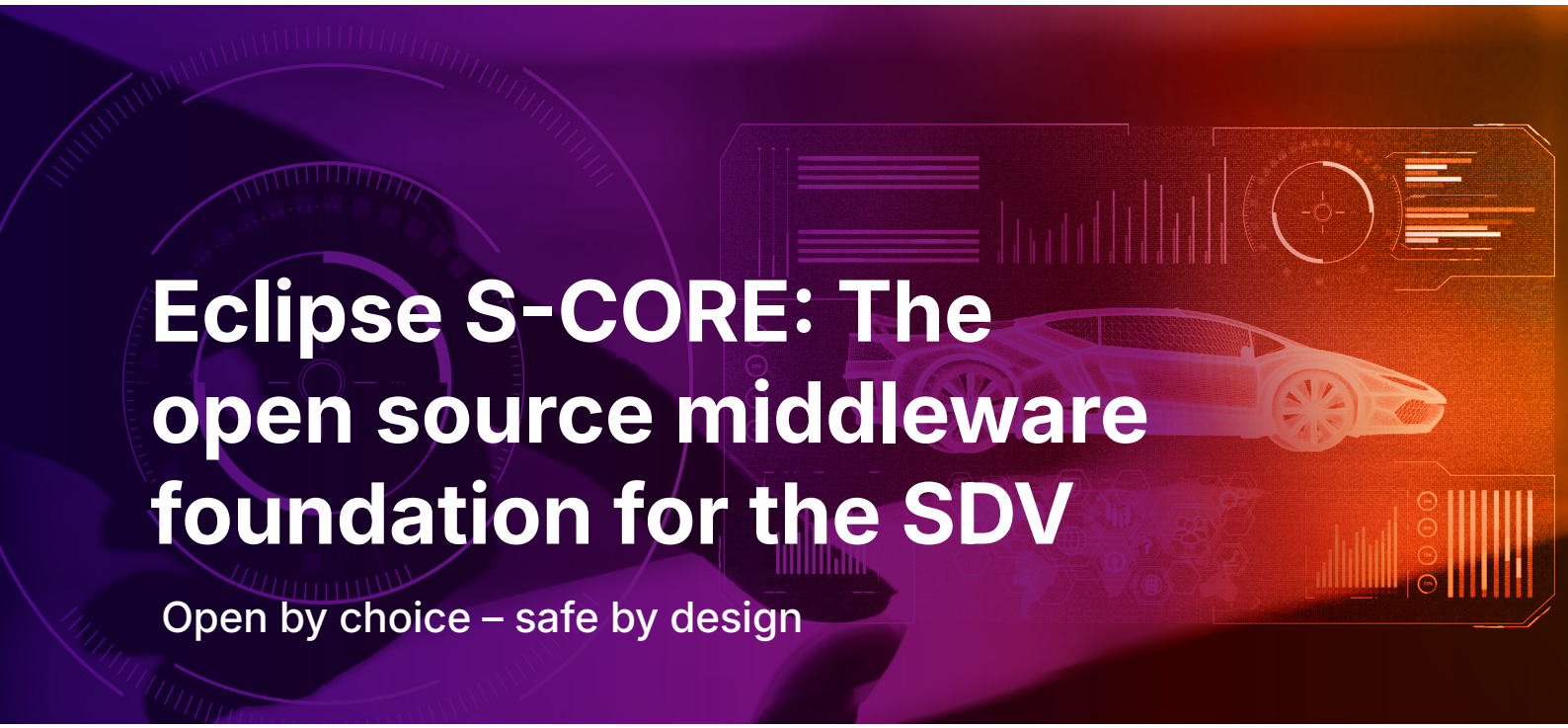
Key takeaways

→ **The Application Framework simplifies SDV software development** through a unified, open application layer.

→ **It decouples applications from OS and middleware**, reducing integration complexity and vendor lock-in.

→ **Automation and code generation accelerate development** in both, prototyping and production setups.

→ **The open, text-based workflow enables agile, collaborative, and AI-assisted development.**



Eclipse S-CORE: The open source middleware foundation for the SDV

Open by choice – safe by design

BY BJÖRN REISTEL

Modern vehicle software complexity has outpaced traditional automotive E/E architectures. Automakers currently spend immense resources engineering foundational functions and restarting development from scratch with each vehicle generation. This approach creates integration bottlenecks and repeated development efforts for every new program.

The Eclipse S-CORE project, developed collaboratively by OEMs, Tier-1 suppliers, and software partners within the Eclipse SDV Working Group, solves this by providing a modular, safety-capable, open source middleware foundation. By focusing on shared code, S-CORE eliminates redundant groundwork and accelerates overall development velocity. The project is developed across several repositories under the Eclipse S-CORE GitHub organisation.[79]

Architectural blueprint & technical core

Eclipse S-CORE acts as the runtime and communication layer between the underlying safe POSIX operating system (e.g., Linux, QNX) and the vehicle's application frameworks.

The technical foundation is built on three core architectural pillars:

- **Low-latency communication and discovery**: S-CORE implements a zero-copy, shared-memory Inter-Process Communication (IPC) framework. This reduces CPU overhead for data-heavy workloads like ADAS. Designed to coexist with and complement industry standards like AUTOSAR Adaptive and COVESA, S-CORE serves as an optimised data transport layer. This allows developers to accelerate critical data paths while maintaining alignment with established automotive software architectures and vehicle data models.

[79] https://github.com/eclipse-score

- **Safe-by-design approach:** S-CORE supports safety-critical applications through a modular design that decouples safety-critical execution paths from non-safety routines. Development processes are regularly audited by external safety bodies to ensure ISO 26262 compliance..
- **Language pragmatism**: S-CORE supports a dual-track programming language strategy. It utilises C++ to preserve compatibility with existing automotive ecosystems and legacy code, while adopting Rust for newly designed, safety-critical middleware components where memory and thread safety are paramount.

Technical deep dives

LoLa communication (eclipse-score/communication)

- Zero-copy data sharing: To handle high-throughput sensor data (e.g., LiDAR, camera images) in ADAS, LoLa utilises physical shared memory segments. Processes read and write directly to these shared memory pools, completely avoiding any data serialisation or copy operations across process boundaries.
- Brokerless service discovery: Eclipse S-CORE implements a lightweight, peer-to-peer flag-file discovery mechanism. Services announce their availability by generating static "flag files" within predefined, system-wide directories. Clients scan these directories to resolve connection details and establish direct peer-to-peer IPC channels.

Safe-by-design approach

- ASIL-B process integrity: S-CORE integrates functional safety standards directly into its development lifecycle. Its processes are continuously audited to ensure systematic capability up to ASIL-B. Crucially, the project delivers safety artifacts along the V-Model, simplifying safety certification for commercial products built on Eclipse S-CORE.
- Software isolation: The modular architecture provides a clean separation between software components. This ensures that non-critical application loops do not interfere with safety-related communications, laying a robust foundation for system-level safety integration.

Dual-language interoperability

- Hybrid codebase integration: S-CORE integrates memory-safe Rust modules with legacy C++ ecosystems. This approach is exemplified within the LoLa framework, where Rust is used to provide safe API bindings around a high-performance C++ core.
- Zero-overhead FFI boundary: By utilising modern interoperability tools (such as the cxx crate), the project avoids translation-layer penalties. The C++ and Rust boundary is bridged using zero-overhead Foreign Function Interfaces (FFI), sharing data structures directly across memory boundaries without performance loss.

The "code-first" paradigm and business value

Eclipse S-CORE follows the "code-first" methodology. Under this collaborative model, the shared, running codebase serves as a primary reference point alongside specifications, ensuring that architectural concepts are instantly runnable and verifiable.

But the project extends far beyond the implementation layer. Eclipse S-CORE delivers not only source code, but a complete set of engineering artifacts. Requirements are captured and linked directly within the repositories, comprehensive tests and automated validation pipelines are established, and detailed technical documentation is maintained alongside the code. This comprehensive approach ensures transparency, safety alignment, and a seamless path to integration.

This model delivers significant economic and strategic advantages:

- Parallel development: Because the stack is runnable down to the OS layer, application developers can download the code and build

brand-differentiating software immediately, without waiting for hardware or speculative software deliveries.

- Validation of deltas: Rather than re-validating the entire middleware stack from scratch for each project, OEMs and integration partners only validate the "deltas" (specific modifications), drastically reducing validation cycles and costs – the single largest time sink in automotive E/E.
- Industrialisation model: Designed to be capable of safety certification, S-CORE's development processes and artifacts are audited by external safety bodies like Exida from day one. Commercial distributors can take this safety-capable stack, customise it for specific target hardware, and deliver a certified, production-ready product, transforming the business model from proprietary licensing to continuous delivery and integration services.

Conclusion

Eclipse S-CORE demonstrates that open source can be more than a development model. It can serve as the foundation for the next generation of SDVs.

By providing a modular, safety-capable middleware stack, embracing a code-first methodology, and enabling collaboration across OEMs, suppliers, and technology partners, the project addresses many of the structural challenges that have historically slowed automotive software development.

As vehicle architectures continue to consolidate around high-performance compute platforms, the value of a shared, production-ready middleware foundation will only increase. Eclipse S-CORE reduces duplicated engineering effort, shortens integration and validation cycles, and allows manufacturers to focus on the software that differentiates their products rather than rebuilding common infrastructure. In doing so, it lowers the barrier to entry, transforming the modern vehicle into an open, accessible, and compelling software platform for innovation.

Ultimately, S-CORE shifts the industry toward a collaborative ecosystem, future-proofing vehicle architectures and establishing open source as the norm in automotive E/E development.

Interested in the project, the full documentation, or getting involved as a contributor? Visit our website.[80]

Key takeaways

→ **S-CORE provides an open, safety-capable middleware foundation** for SDVs.

→ **Its modular, code-first architecture accelerates development** and reduces integration effort.

→ **Built for high-performance SDVs**, it supports deterministic execution and low-latency communication.

→ **Open collaboration reduces costs** and enables faster automotive innovation.

[80] https://eclipse.dev/score



Eclipse ThreadX

The software engine for SDVs

BY FRÉDÉRIC DESBIENS

As automotive architectures undergo a fundamental shift toward software-defined designs, traditional vehicle wiring and electrical setups are being replaced by centralised zonal architectures. In this model, high-performance central computers handle complex logic, while localised zonal controllers aggregate inputs and manage physical subsystems. However, this transition to centralised compute does not mean that microcontrollers (MCUs) are vanishing from the vehicle. On the contrary, MCUs remain essential at the edge of the network, acting as resilient gateway nodes, sensor aggregators, and actuator drivers. Within these zonal controllers, Real-Time Operating Systems (RTOSes) continue to play a vital role, ensuring the deterministic execution, low latency, and safety-critical reliability required to keep vehicles running safely.

Furthermore, the economics of embedded software development are shifting. Open source RTOSes are increasingly displacing the traditional, proprietary commercial operating systems that historically dominated the automotive and industrial sectors. Historically, OEMs paid steep licensing fees and royalties for proprietary RTOS solutions to guarantee functional safety. Today, the neutrality, community-driven innovation, and transparency of open source projects – backed by robust entities like the Eclipse Foundation – provide safety-certified guarantees without vendor lock-in. This open model enables rapid prototyping, collaborative security auditing, and a shared maintenance burden, establishing OSS as the new baseline for next-generation vehicle architectures.

The RTOS landscape: Finding the perfect match

In the embedded and IoT space, selecting the right Real-Time Operating System (RTOS) is as critical as choosing the right silicon. While desktop and server developers have standardised on Linux, the resource-constrained microcontroller market remains fragmented. Embedded system designers must navigate a complex array of constraints: memory foot-

print, scheduling determinism, integration complexity, licensing neutrality, and safety certification.

To understand the unique value proposition of Eclipse ThreadX,[81] it is helpful to contrast it with the other major open source contenders in the market: FreeRTOS and Zephyr RTOS.

- **FreeRTOS**: Controlled by Amazon Web Services (AWS), FreeRTOS is a minimalist kernel providing basic scheduling primitives, queues, and task management. It has a massive user base and an ultra-low memory footprint. However, most advanced functionality (such as TCP/IP networking, file systems, and USB stacks) is left to third-party libraries or hardware vendor-specific SDKs. The integration effort and risk fall squarely on the developer, often resulting in fragmented, vendor-locked codebases.
- **Zephyr Project**: Maintained as a collaborative project under the Linux Foundation, Zephyr is a comprehensive OS platform. It includes a built-in device driver model, security features, and a rich, highly configurable networking stack. While powerful, Zephyr is complex, has a steep learning curve, and demands a significantly higher baseline memory footprint. It is rarely suited for the smallest, most resource-constrained microcontrollers. Additionally, while the project has been pursuing safety certification, it is not explicitly targeting the ISO 26262 standard directly relevant to automotive.
- **Eclipse ThreadX**: Positioned as a neutral, vendor-independent project under the Eclipse Foundation, ThreadX offers a best-in-class middle ground. It is delivered not as a bare kernel, but as a cohesive, pre-validated middleware stack built around an efficient "picokernel" architecture. It combines the low footprint of a minimalist kernel with the integrated, production-ready features of a full platform (see ThreadX architecture in **figure 1**).

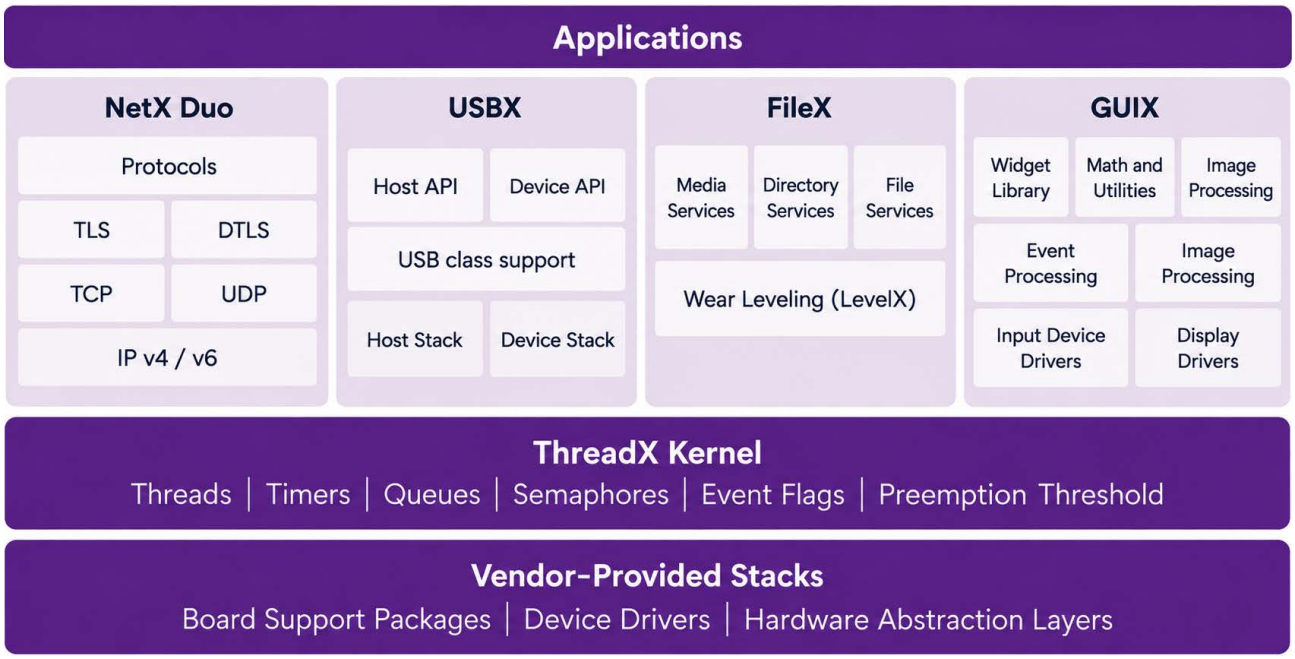


Figure 1: ThreadX architecture

Ultimately, ThreadX provides a highly deterministic runtime environment with minimal integration overhead. This makes it a preferred choice for professional and enterprise-grade embedded deployments, as evidenced by its presence in over 12 billion devices worldwide.

[81] https://github.com/eclipse-threadx/threadx

Technical differentiators: What makes ThreadX stand out

Under the hood, ThreadX implements several advanced scheduling and resource management mechanisms that set it apart from traditional real-time kernels.

Deterministic picokernel architecture

ThreadX is designed from the ground up for hard real-time environments where latency must be bounded and predictable. Its context-switching paths are heavily optimised in assembly language, minimising CPU overhead. Unlike typical RTOS kernels that use a tick-based scheduler with high jitter, the ThreadX scheduling engine is event-driven and strictly priority-preemptive, ensuring rapid interrupt response and deterministic execution.

Preemption-threshold scheduling

A key innovation in ThreadX is its unique **preemption-threshold** mechanism. In a standard priority-preemptive scheduler, any thread of higher priority will instantly interrupt a running thread of lower priority. While this guarantees responsiveness, it can lead to excessive context-switching overhead and priority inversion issues.

Preemption-threshold allows a developer to specify a priority range within which a thread cannot be preempted by other, slightly higher-priority tasks. For example, a thread running at priority 10 can set its preemption-threshold to priority 5. This means that while threads with priorities 1 to 4 can still preempt it, threads with priorities 5 to 9 are blocked from doing so. This enables fine-grained control over scheduling, allowing developers to group related tasks, eliminate unnecessary context switches, and reduce the need for synchronisation primitives like semaphores. Furthermore, it helps minimise stack memory requirements, as threads that share a preemption-threshold cannot execute concurrently, allowing them to potentially share memory resources.

Tightly integrated middleware stack

Unlike FreeRTOS, where developers must cobble together networking and file system libraries, ThreadX provides a suite of pre-validated middleware designed to work together seamlessly:

- **NetX Duo**: A high-performance, dual-stack IPv4 and IPv6 TCP/IP networking stack.
- **FileX**: A robust FAT12/16/32 and exFAT-compatible file system featuring fault-tolerance capabilities.
- **USBX**: A comprehensive USB host and device stack supporting a wide range of device classes.
- **GUIX**: A modular graphics and user interface library designed for embedded displays.
- **LevelX**: A wear-levelling library that manages raw flash memory under FileX.

Because these components are co-designed, they eliminate the integration risk and testing overhead that plague other RTOS ecosystems.

ThreadX in the SDV and OpenHW integration

As the automotive industry transitions to SDVs, the need for high-performance, safety-critical real-time processing has never been greater. Modern automotive architectures consolidate multiple vehicle functions onto powerful System-on-Chips (SoCs). This requires mixed-criticality execution, where safety-critical tasks (such as body control, ADAS sensor processing, or powertrain management) must execute reliably on the same chip as non-critical workloads (like infotainment or telematics).

ThreadX is uniquely positioned for the SDV ecosystem due to its extensive safety credentials. It is pre-certified for functional safety standards, including ISO 26262 ASIL-D (automotive), IEC 61508 SIL 4 (industrial), and EN 50128 SIL 4 (rail).

This safety pedigree matches perfectly with the open source silicon initiative led by the OpenHW

Foundation. By running a certified RTOS like ThreadX on verified, open source processor cores like the OpenHW CVA6 or the CV32E40P, automotive designers can achieve full stack visibility – from the application software down to the literal logic gates of the silicon.

RISC-V highlights in recent releases

The alignment between ThreadX and the RISC-V ecosystem has accelerated rapidly, culminating in the **v6.5.1 release** in June 2026. This release showcases several standout contributions from the community:

- **Advanced BSP Support**: 10XEngineers contributed a Board Support Package (BSP) for the Banana Pi BPI-F3, which features the SpacemiT K1 octa-core RISC-V SoC. They also introduced QEMU-based RISC-V continuous integration (CI) to ensure the stability of the codebase.
- **Vector Math Acceleration**: Alibaba contributed RISC-V Vector Extension (RVV) support for 64-bit architectures, enabling ThreadX tasks to leverage hardware-accelerated vector instructions for edge AI and signal processing workloads.
- **Silicon Integration**: The Eclipse Foundation refactored and consolidated the RISC-V port, adding a new port for the **CORE-V MCU** (integrating the CV32E40P[82]).

Meanwhile, Quintauris – the joint venture founded by major semiconductor players Bosch, Infineon, Nordic, NXP, and Qualcomm to drive RISC-V adoption – contributed a new 32-bit port for the OpenHW CVA6[83] application-class processor. This contribution signals a major industry commitment to deploying ThreadX as a standard runtime for next-generation automotive silicon platforms.

The momentum continued with v6.5.2 in October 2026. Afonso Oliveira from Ainekko contributed a board example for Erbium, the 16-hart RISC-V platform of the OpenHW Foundation's CORE-ET project. Ainekko contributed CORE-ET to OpenHW, and it pairs many-core 64-bit RISC-V compute with MRAM for low-power inference at the edge. The example has been verified on the Erbium simulator, since silicon is expected in December 2026. Wei-Chen Lai added lazy FPU stacking to the RV64 port, so threads that never use floating point no longer pay to save and restore its registers. Akif Ejaz, a committer from 10xEngineers, brought the RV32 and RV64 ports into line with the RISC-V specification and turned on the RISC-V regression suite in CI against QEMU targets. The suite now also runs at -O2. The Eclipse Foundation also fixed a context-save defect that sent interrupts back to the wrong address when execution-change notification was enabled.

The road ahead: A RISC-V-first RTOS

The roadmap for ThreadX on RISC-V aims to deliver an autonomous, highly secure stack without the overhead of heavy virtual memory operating systems:

- **Physical Memory Protection (PMP)**: ThreadX is implementing native support for RISC-V PMP. This allows the kernel to protect itself, implement stack overflows/canaries, and enforce task-level memory isolation directly in Machine-mode (M-mode), avoiding the performance penalties of Supervisor-mode (S-mode) page tables.
- **S-Mode and SMP Ports**: For larger application cores like the CVA6, ThreadX is developing a Supervisor-mode port and Symmetric Multiprocessing (SMP) support. This configuration leverages OpenSBI in M-mode to coordinate task execution across multiple cores.

[82] https://github.com/openhwgroup/cv32e40p

[83] https://github.com/openhwgroup/cva6

- **ZoneX Hypervisor Integration**: ZoneX is a thin partitioning hypervisor designed to run multiple isolated ThreadX instances on a single SoC. It provides hard real-time latency guarantees and a certified safety path, enabling clean separation of mixed-criticality workloads in SDV architectures.

Conclusion: The foundation of a trustable stack

Eclipse ThreadX is no longer just a standalone kernel. By combining its proven, safety-certified runtime engine with a collaborative, vendor-neutral governance model at the Eclipse Foundation, it has become a cornerstone of the open source embedded ecosystem.

When paired with open hardware like the OpenHW Foundation's CORE-V processor cores, ThreadX enables a fully transparent, trustable, and scalable stack. For the future of the SDVs and the broader IoT landscape, this hardware-software co-design ensures that safety, security, and performance are built into the very fabric of the system.

Key takeaways

→ **ThreadX provides a lightweight, safety-certified RTOS** for SDVs.

→ **Its deterministic architecture delivers high performance** for safety-critical automotive systems.

→ **Native RISC-V and OpenHW support** enables an open, transparent software stack.

→ **ThreadX is a key building block for trusted, next-generation SDV platforms**.

Open source silicon: The new frontier of software-hardware co-design

The code in the copper: When software meets RTL

BY FRÉDÉRIC DESBIENS

For decades, the Eclipse Foundation was synonymous with software. The hardware underneath was treated as a given – a black box of copper and silicon delivered by chipmakers, accompanied by a PDF datasheet. Porting an operating system or writing low-level drivers meant wrestling with reference manuals that are thousands of pages long. These documents are frequently outdated, ambiguous, and contain subtle errata discovered only through gruelling hours of trial and error in the lab. The boundary between hardware and software teams was a thick, impenetrable wall.

That wall began to crumble when the OpenHW Group[84] transitioned to become the **OpenHW Foundation (figure 1)**, an Eclipse Foundation Working Group. This transition brought open source hardware into the same cooperative governance model and collaborative ecosystem that has powered open source software for decades. It was a structural shift, but it also unlocked a technological gamechanger that is reshaping how we write software for embedded platforms.

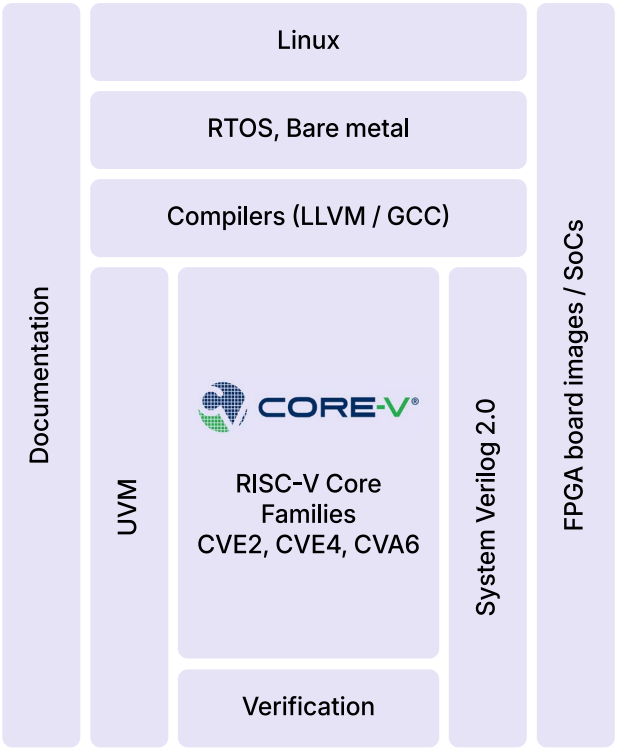


Figure 1: The scope of OpenHW Foundation work

[84] https://openhwfoundation.org/

In the era of AI-assisted development, large language models (LLMs) and agents are helping developers write code at unprecedented speeds. However, when writing a driver or porting an operating system to a proprietary, closed-source chip, the AI is severely limited. It can only draw from public header files. It cannot verify how the hardware actually behaves because the underlying Register Transfer Level (RTL) description of the silicon is a closely guarded corporate secret.

This limitation disappears when the hardware itself is open source. Because the OpenHW Foundation's CORE-V processor cores are defined in open SystemVerilog RTL on GitHub,[85] developers can feed the literal silicon implementation directly into the AI's context window.

The author of this chapter experienced this while porting the safety-certified Eclipse ThreadX RTOS (see page 56) to the OpenHW CORE-V MCU.[86] Porting an RTOS requires precise alignment with the chip's interrupt controller and startup vector tables. Instead of guessing based on draft PDFs, the author provided the SystemVerilog RTL files of the interrupt controller directly to Copilot (**figure 2**).

The AI did not just read the headers; it read the literal hardware description code. It analysed the logic gates, mapped the interrupt lines, and generated the exact assembly startup code and C structures required to wire the ThreadX interrupt vector table.

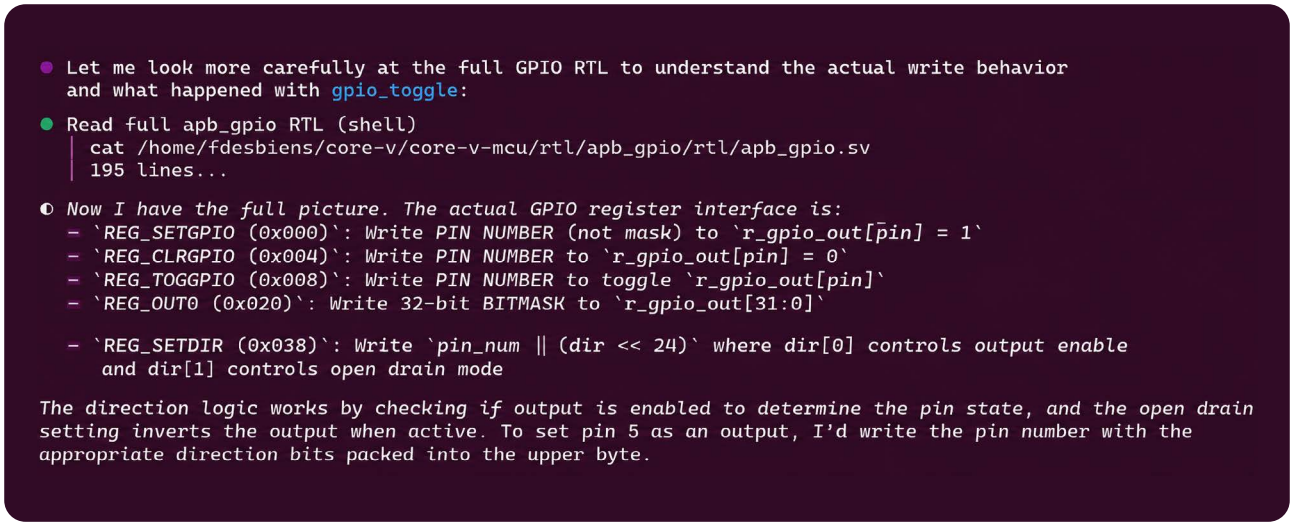


Figure 2: Copilot reading RTL

This represents a fundamental shift in software-hardware co-design. For the first time, software developers and AI assistants can peer directly into the copper to write software that matches the silicon exactly. It proves that when hardware is open, software enablement becomes faster, safer, and infinitely more robust.

Re-engineering sovereignty: The value of open source hardware for Europe's SDV ecosystem

As the automotive industry transitions toward SDVs, vehicular electronics complexity is exploding. Modern cars are high-performance computers on

[85] https://github.com/openhwgroup

[86] https://github.com/openhwgroup/core-v-mcu

wheels. This shift presents Europe with both a major economic opportunity and a critical strategic vulnerability. European OEMs and Tier-1 suppliers must innovate rapidly while maintaining strict standards for functional safety, cybersecurity, and supply chain resilience.

By using open source cores and accelerators as building blocks, the European automotive ecosystem can avoid reinventing foundational microarchitectures. Instead of designing basic processors from scratch, industry can focus investment where it creates the most competitive value: advanced verification, tooling, packaging, domain-specific acceleration (AI and ADAS workloads), software enablement, safety certification, and deployment.

Open source silicon is therefore a force multiplier for Europe's semiconductor ambitions, supporting sovereignty without fragmentation. Instead of building isolated proprietary solutions or relying on closed external IP, European firms can collaborate on shared, highly verified hardware foundations. This strategy supports resilience without isolation, letting companies secure supply chains and retain IP rights while remaining integrated with a global ecosystem.

This open approach is vital from an SDV perspective. In an SDV, software and hardware must be co-designed to support mixed-criticality workloads, where safety-critical functions (like braking or steering) run alongside non-critical systems (like infotainment) on the same system-on-chip (SoC). This requires robust virtualisation and strict memory protection.

When the underlying processor cores and interconnects are open source, software engineers have full visibility into the silicon's behaviour, making it far easier to design secure hypervisors. Reusable hardware foundations reduce integration risks, speed up ISO 26262 safety certification, and significantly accelerate time-to-market.

Furthermore, open collaboration around RISC-V hardware supports workforce development. European universities can contribute to and train students on commercial-grade, silicon-proven designs. Graduates enter the workforce with hands-on experience in the exact technologies used by industry, bridging the gap between academic research and commercial economic impact.

The OpenHW lineup: From verified cores to platform enablers

The OpenHW Foundation has established a rigorous verification pipeline to ensure that its open source designs are ready for commercial silicon fabrication. Its lineup is divided into families tailored to different computing needs, from low-power microcontrollers to high-performance application processors, and is now expanding into complete silicon platforms.

The CORE-V family

The CORE-V family represents the core processor IP developed and verified by the OpenHW Foundation.

- **CVE2[87] (CORE-V Embedded 2-stage)**: Small, ultra-low-power, 32-bit RISC-V cores with an efficient 2-stage pipeline. Optimised for a minimal silicon footprint, they serve as controllers in sensor nodes and deeply embedded subsystems where energy conservation is paramount.
- **CVE4 (CORE-V Embedded 4-stage)**: Originally based on the PULP RI5CY core, these 32-bit, 4-stage, in-order RISC-V cores target compute-intensive applications. Key variants include:
  - **CV32E40P**:[88] Supports DSP and SIMD extensions. It powers the CORE-V MCU[89] project, integrating the core with peripherals like I^2C, SPI, UART, and PWM onto a physical development kit.

[87] https://github.com/openhwgroup/cve2

[88] https://github.com/openhwgroup/cv32e40p

[89] https://github.com/openhwgroup/core-v-mcu

  - **CV32E40X**:[90] Features a coprocessor extension interface, allowing designers to attach custom hardware accelerators without modifying the base core.
  - **CV32E40S**:[91] A security-focused variant with dual privilege modes, physical memory protection (PMP), and anti-tampering features.

- **CVA6**[92] **(CORE-V Application 6-stage)**: Originally known as the PULP Ariane core, the CVA6 is an application-class, 6-stage, single-issue, in-order processor core implementing the RV32GC or RV64GC extensions. It features three privilege levels (Machine, Supervisor, and User), making it fully capable of running a Unix-like operating system such as Linux.
- **CV-Wally**:[93] A highly configurable, 5-stage pipelined RISC-V core. As the primary reference implementation in the landmark textbook **RISC-V Microprocessor System-on-Chip Design**,[94] CV-Wally is poised to shape the careers of the next generation of chip designers, providing students with a direct bridge from academic theory to production-ready SystemVerilog silicon design.

Recent developments for CVA6

The CVA6 family has seen significant advancements aimed at meeting the demanding requirements of industrial and automotive applications. A key development is the implementation of **dual-core lockstep configurations**, which are essential for achieving the highest levels of functional safety (ASIL-D) under the ISO 26262 standard. In a lockstep configuration, two CVA6 cores execute the same instruction stream in parallel, and their outputs are compared cycle-by-cycle to detect and mitigate transient hardware faults caused by radiation or electrical noise.

In addition to safety enhancements, the OpenHW ecosystem is pushing the boundaries of architectural research. At RISC-V Summit Europe 2026, a new **128-bit CVA6 variant** was presented. This work was discussed in a talk titled "A Proof-of-Concept RISC-V with 128-bit Extension" by Frédéric Pétrot, highlighting the work of team **MADMAX** (a joint effort of Université Grenoble Alpes, Inria, and CNRS). While this 128-bit design remains a research project that is still a long way from commercial fabrication, it serves as a testament to the power of open source silicon. Open source licensing makes it possible for academic and research teams to cooperatively explore the architectural implications and potential of massive memory address spaces long before the physical hardware is cast in silicon.

The CORE-ET Silicon Platform

The introduction of the **CORE-ET (CORE Emerging Technologies) Silicon Platform** project marks a new era for the OpenHW Foundation. Previously, the foundation's primary mission was to deliver and verify individual processor IP cores. While verified, open source processor IP remains a cornerstone of the foundation's work, the CORE-ET project represents an ambition to deliver a complete ecosystem of RISC-V platform-enabling components.

Specifically, the CORE-ET platform establishes a unified, pre-verified System-on-Chip (SoC) framework. Rather than forcing designers to build their own bus fabrics and peripheral bridges, CORE-ET provides a standardised interconnect (leveraging AMBA AXI4 and APB protocols) linking the processor cores to high-bandwidth memory controllers (sup-

[90] https://github.com/openhwgroup/cv32e40x

[91] https://github.com/openhwgroup/cv32e40s

[92] https://github.com/openhwgroup/cva6

[93] https://github.com/openhwgroup/cvw

[94] https://shop.elsevier.com/books/risc-v-system-on-chip-design/harris/978-0-323-99498-9

porting LPDDR4 and DDR4), standard interfaces (like CAN-FD, PCIe, and Gigabit Ethernet), and a RISC-V debug module.

A centrepiece of this initiative is the CORE-ET AI accelerator, which integrates a dedicated neural processing unit (NPU) optimised for matrix multiplication and low-power tensor operations (supporting INT8 and FP16 data types). By standardising these interfaces, OpenHW reduces the 'interconnect tax' and validation complexity, allowing developers to focus on custom hardware acceleration.

Conclusion: The roadmap to open SDV silicon

Ultimately, the software-defined vehicle demands a computing paradigm that traditional, siloed proprietary hardware vendors can no longer sustain. The tight feedback loop required between safety-critical software stack developers, virtualisation architectures, and underlying silicon demands absolute transparency.

By providing commercially verified, open source processor IP and pre-integrated silicon platforms, the OpenHW Foundation is positioned to be the future of SDV silicon. It provides the only viable path to achieve true European technology sovereignty, robust supply chain resilience, and rapid, safety-certified innovation. In the future of automotive design, the road will be defined by software, but it will run on open silicon.

Key takeaways

→ **Open silicon enables better AI-assisted software development** by giving developers and AI full visibility into hardware.

→ **OpenHW accelerates SDVs** through transparent, reusable RISC-V platforms.

→ **Open hardware improves safety, security, and certification** while reducing development time.

→ **Open collaboration strengthens Europe's semiconductor sovereignty and innovation ecosystem.**



An open development toolchain to support SDV stack development

Tool considerations and recommendations

BY SARA GALLIAN

The transition to software-defined vehicles changes not only vehicle architecture but also the way automotive software is designed, developed, integrated, and validated. Growing complexity, shorter development cycles and continuous updates require engineering environments that connect previously separate disciplines. A collaborative framework therefore needs more than reusable in-vehicle components: it also requires an integrated toolchain spanning the entire software lifecycle.

The broader Eclipse Foundation ecosystem provides many of the building blocks needed to create this toolchain. Instead of prescribing a monolithic environment, it offers interoperable platforms that organisations can combine according to their processes and domains. Together, these technologies support architecture, modelling, implementation, integration, testing, deployment, and analysis across the SDV stack.

Why open governance matters

Automotive development environments have traditionally relied heavily on proprietary tools. Although these may provide advanced domain-specific capabilities, they can create vendor lock-ins, restrict integration and make workflows difficult to adapt. In an SDV ecosystem, where software from numerous organisations, both proprietary and open source, must work together over a long vehicle lifecycle, these limitations become increasingly significant.

An open source toolchain provides an alternative infrastructure. Transparent interfaces and extension mechanisms allow organisations to adapt tools, connect them to existing systems, and collaborate on non-differentiating capabilities. Companies can still develop proprietary extensions, products, and services on top.

However, publishing source code alone does not automatically create a sustainable ecosystem. A single-vendor project may remain dependent on the strategy of one organisation. Vendor-neutral governance under an open source foundation provides transparent rules for participation, decision-making, intellectual property management and project evolution. It also helps ensure that the toolchain can evolve with changing architectures, standards, and regulatory requirements.

A common build and tooling foundation

Eclipse SDV Hephaestus[95] addresses a fundamental challenge within an open, modular ecosystem: individual projects have often created their own build and tooling environments. This duplicates effort and makes it harder to integrate several components into a complete solution.

Hephaestus is intended to harmonise these non-differentiating capabilities across Eclipse SDV. It provides a common foundation for build systems, development environments, CI/CD workflows, and related infrastructure. Shared conventions and reusable configurations can reduce the time projects spend solving the same tooling problems independently.

This role is particularly important for a full-stack framework. A collection of compatible runtime components does not automatically form an integrated stack if each component uses different build procedures, dependency models, or validation pipelines. Hephaestus helps bridge this gap by making joint builds and integration workflows more consistent and reproducible. It therefore acts as a connective layer between individual Eclipse SDV projects and the broader development environment.

Tailored engineering environments

Eclipse Theia[96] provides the user-facing foundation for modern cloud and desktop development environments. It can serve as a general-purpose IDE, but it can also be used to create specialised automotive tools combining source code, graphical models, documentation, terminals, dashboards, and domain-specific workflows.

Its modular architecture allows tool providers to select the capabilities required for a particular use case. An application development environment may focus on coding, debugging, container management, and deployment. A systems engineering environment may integrate architecture editors, requirements, model validation, and traceability. Both can be built on the same platform.

The **Open VSX Registry**[97] provides access to extensions for common development tasks, while organisations remain free to create private or automotive-specific extensions. Theia-based environments can run locally or in the cloud. Cloud deployment makes it easier to provide consistent, reproducible workspaces in which dependencies, configurations, and tool versions are managed centrally. This is particularly valuable for distributed programs involving several organisations.

Reusable Eclipse Foundation building blocks

The Eclipse Foundation ecosystem contains complementary technologies for constructing an integrated SDV engineering environment (see table below).

[95] https://projects.eclipse.org/projects/automotive.hephaestus

[96] https://theia-ide.org/theia-platform/

[97] https://open-vsx.org/

Building block	Role in the toolchain
Eclipse SDV Hephaestus	Common build, development tooling and CI/CD foundation
Eclipse Theia	Platform for desktop and browser-based engineering environments
Open VSX Registry	Open distribution channel for IDE extensions
Eclipse GLSP™	Framework for browser-based graphical model editors
Eclipse Sprotty	Visualisation of diagrams and structured engineering information
Langium	Development of textual domain-specific languages
EMF.cloud	Cloud-native model management and model-based services
Theia AI	Integration of configurable AI capabilities into engineering tools
Eclipse LMOS	Development and operation of interoperable multi-agent systems
Eclipse Enclave	Executes autonomous agents within controlled and auditable sandboxes
Eclipse Papyrus	Standards-based UML, SysML and MARTE modelling environment
Eclipse Capella	Model-based design of system, software and hardware architectures
Eclipse SSAM	Secure, lightweight container execution on automotive ECUs

These are not simply isolated end-user applications. They form a collection of components from which organisations can construct domain-specific tools and workflows. Open interfaces allow individual elements to be extended or replaced without redesigning the entire environment.

Connecting code and models

A full-stack SDV toolchain must support both code-centric and model-based development. Vehicle architectures include logical functions, software components, services, communication relationships, deployment targets, and hardware resources. Engineers need to move between these perspectives without maintaining disconnected representations in different tools.

Eclipse Papyrus[98] provides an integrated environment for creating and editing EMF-based models, with particular support for UML, SysML and MARTE. Within an SDV toolchain, it enables engineers to model software structures, system behaviour and relationships between vehicle functions, and technical components using established modelling standards. Its customisation mechanisms also allow organisations to adapt modelling environments to their own automotive methods and domain-specific requirements.

Eclipse Capella[99] (see page 72) is an extensible, field-proven model-based systems engineering tool for designing system, software and hardware architectures. Based on the Arcadia methodology, it supports engineers from operational and system analysis through logical and physical architecture design. In a full-stack SDV framework, Capella can help connect vehicle-level needs and functions with the software and hardware architectures required to implement them.

Eclipse GLSP™,[100] Sprotty,[101] Langium,[102] and EMF Cloud[103] provide foundations for graphical and textual modelling. A graphical architecture view can show components, interfaces, data flows, and deployment relationships. A domain-specific language, such as the Kotlin-based architecture approach presented in the toolchain, can provide a concise, version-controllable representation of the same information.

This combination is valuable for collaborative engineering. Textual artefacts work well with Git-based workflows, automated checks and code reviews, while diagrams make complex relationships easier to understand. When both views use a shared underlying model, teams can choose the most appropriate representation without creating conflicting sources of truth.

AI-supported engineering

Artificial intelligence is becoming another layer of the development environment. **Theia AI**[104] enables developers to integrate AI functionality while retaining control over models, prompts, data access, and user interactions. Instead of relying on a fixed assistant, organisations can build agents for particular engineering tasks and connect them to their preferred infrastructure.

Eclipse LMOS[105] extends this concept from individual assistants to enterprise-ready multi-agent systems. It provides a vendor-neutral platform for developing, deploying, and operating specialised agents in the cloud or on-premises. LMOS supports agent discovery, task routing, collaboration, and lifecycle management. The LMOS Protocol uses concepts from the W3C Web of Things architecture to

[98] http://eclipse.dev/papyrus/

[90] https://mbse-capella.org/

[100] https://eclipse.dev/glsp/

[101] https://projects.eclipse.org/projects/ecd.sprotty

[102] https://projects.eclipse.org/projects/ecd.langium

[103] https://eclipse.dev/emfcloud/

[104] https://theia-ide.org/

[105] https://eclipse.dev/lmos/, see also: https://eclipsesdv.org/blogs/we-dont-believe-that-ai-agents-should-only-live-on-the-platforms-of-a-few-big-companies/

facilitate interoperability among agents, applications, tools, and data sources.

Together, Theia AI and LMOS enable a modular AI toolchain. Theia provides the interface within the engineering environment, while LMOS can coordinate the agents behind it. An engineer could, for example, request an assessment of an architectural change. LMOS could direct parts of the task to architecture, source-code analysis and testing agents, then return their combined findings through the Theia-based tool.

This approach allows organisations to control which information agents can access, which models process it and where those models operate. AI assistance does not replace engineering responsibility. Its purpose is to reduce repetitive work, improve access to relevant information, and help users navigate increasingly complex systems.

Eclipse Enclave[106] provides a sandboxed runtime for autonomous development agents. Each agent operates within an isolated container with its own filesystem, processes, and network stack. Outbound connections can be restricted to approved domains and logged, while separate Git worktrees allow several agents to work on the same project in parallel without interfering with one another. Enclave also provides central controls and audit trails for inspecting agents and reviewing their activities. By separating agent execution from agent identity, it allows organisations to use different agents and editors while applying consistent security, observability and governance policies.

From projects to a continuous workflow

The toolchain delivers its greatest value when its components are connected. Architecture descriptions can inform implementation; software artefacts can be linked to build and deployment configurations; and validation results can be presented in the same environment in which engineers make changes.

Eclipse SSAM[107] is a lightweight container execution framework designed for resource-constrained automotive ECUs. It supports the installation, removal, and updating of container packages, executes them through an OCI-compatible runtime, and continuously verifies their integrity to detect unauthorised modifications. Resource isolation and fast start-up make SSAM particularly relevant for securely deploying containerised applications to in-vehicle targets.

Eclipse Hephaestus can provide common build and CI/CD workflows across Eclipse SDV projects, while Theia brings the development experience together. Modelling technologies connect architectural intent with technical artefacts, and Theia AI and LMOS introduce specialised assistance across design, implementation, and testing. Automation can then evaluate changes continuously instead of waiting for late integration phases.

The open development toolchain is therefore an enabling layer of the full-stack Eclipse SDV framework. It connects people, processes, models, and software components across organisational and technical boundaries. Based on modular platforms, shared infrastructure, open interfaces, and vendor-neutral governance, it turns individual projects into a more coherent development ecosystem for software-defined vehicles. See also **figure 1** for more Eclipse Foundation projects and how they fit into the SDV landscape.

[106] https://projects.eclipse.org/projects/ecd.enclave

[107] https://projects.eclipse.org/projects/automotive.ssam

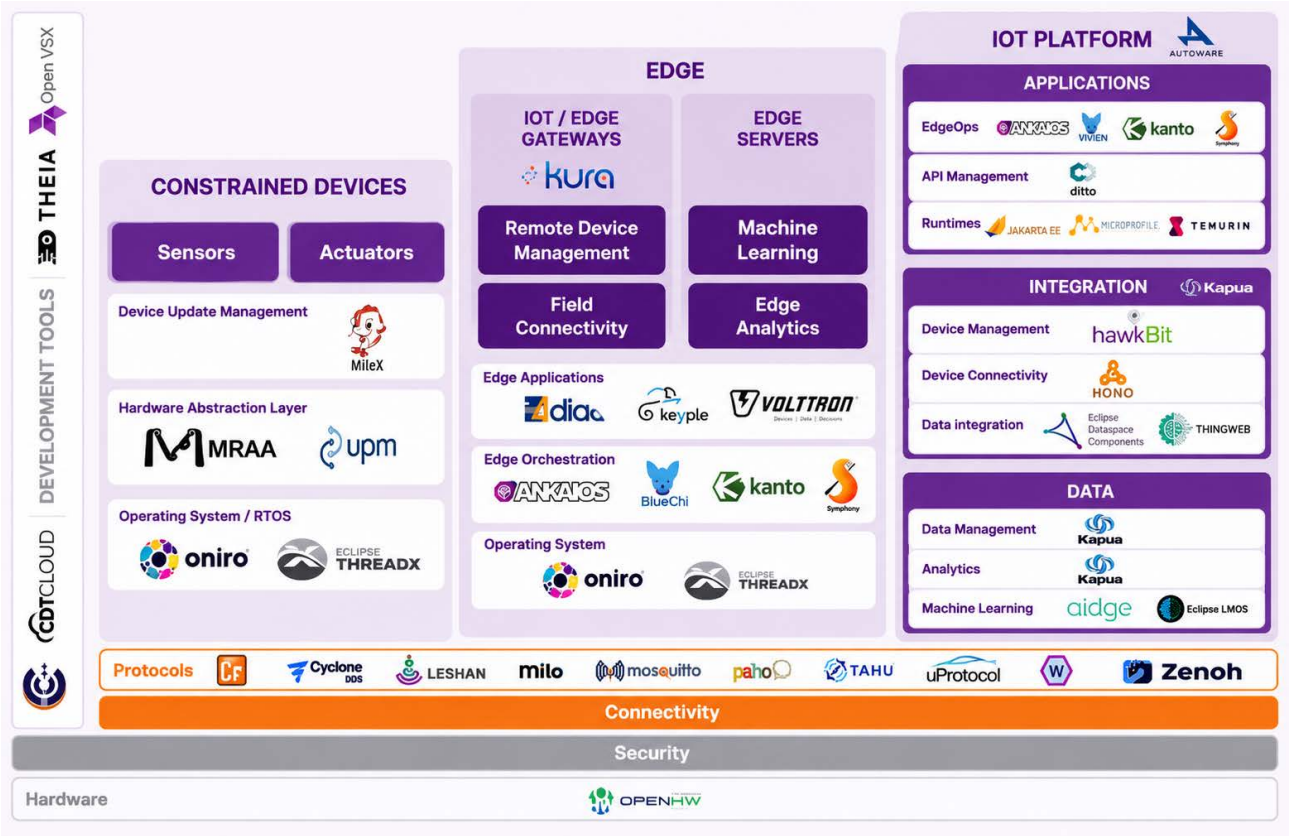


Figure 1: Eclipse projects overview

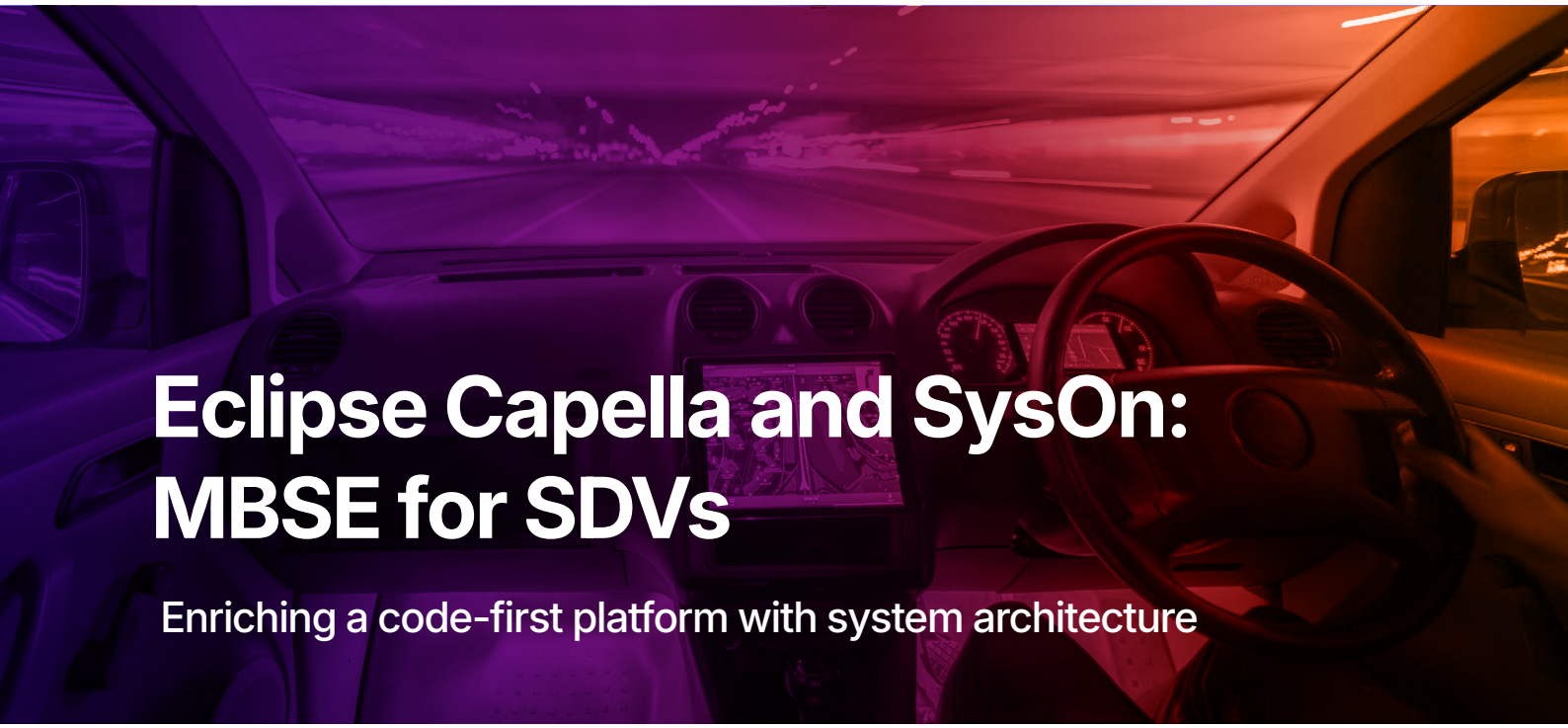
Key takeaways

→ **The Eclipse ecosystem provides a modular, open toolchain** spanning architecture, development, integration, testing, and deployment across the full SDV stack.

→ **Vendor-neutral governance reduces lock-in** and enables automakers, suppliers, and technology providers to collaborate on shared engineering infrastructure.

→ **Eclipse Theia and complementary modelling technologies** connect code-centric and model-based workflows within tailored cloud or desktop environments.

→ **Theia AI and Eclipse LMOS enable configurable AI assistance** and multi-agent collaboration across architecture, coding, testing, and validation.



Eclipse Capella and SysOn: MBSE for SDVs

Enriching a code-first platform with system architecture

BY MARTIN LE BOURGEOIS

The automotive industry is undergoing a profound transformation. Vehicles are increasingly defined by software, data, and connected services rather than mechanical systems alone. This evolution toward SDVs is reshaping how vehicles are designed, developed, and maintained.

Eclipse SDV is one of the leading open source initiatives supporting this transformation. By bringing together automakers, suppliers, and technology providers, it promotes a modular and vendor-neutral ecosystem for next-generation vehicle software platforms.

As software platforms evolve continuously through iterative development cycles, maintaining a consistent understanding of the overall system becomes increasingly challenging. Software, hardware, services, safety constraints, and operational behaviours must evolve together across multiple teams and disciplines.

This is where Model-Based Systems Engineering (MBSE)[108] can provide complementary support.

MBSE as a complement to SDV engineering

Traditional document-based engineering approaches quickly become difficult to maintain in highly iterative SDV environments. Specifications are fragmented across teams and tools, making impact analysis and system consistency harder over time.

MBSE addresses this challenge by introducing system-level models that capture requirements, functions, components, interfaces, and constraints in a consistent way.

These models help engineering teams:

- maintain a global view of the system,
- understand dependencies between components,
- assess the impact of changes,
- support traceability across the lifecycle.

[108] https://mbse-capella.org/what_is_mbse.html

They also provide multiple views adapted to different engineering concerns, from operational analysis to architecture definition and validation (see **figure 1**).

MBSE does not replace code-first SDV approaches. Instead, it complements them by providing a stable architectural perspective while implementations evolve continuously. **Figure 2** illustrates how Capella and SysON are positioned within an SDV development lifecycle.

This is particularly useful during early engineering phases, where key architectural decisions must be made before implementation, but also during iterative development and validation activities.

Using the SDV Companion Application[109] example, the rest of this article illustrates how the Arcadia methodology and the Eclipse Foundation MBSE tools Capella and SysON can support SDV engineering.

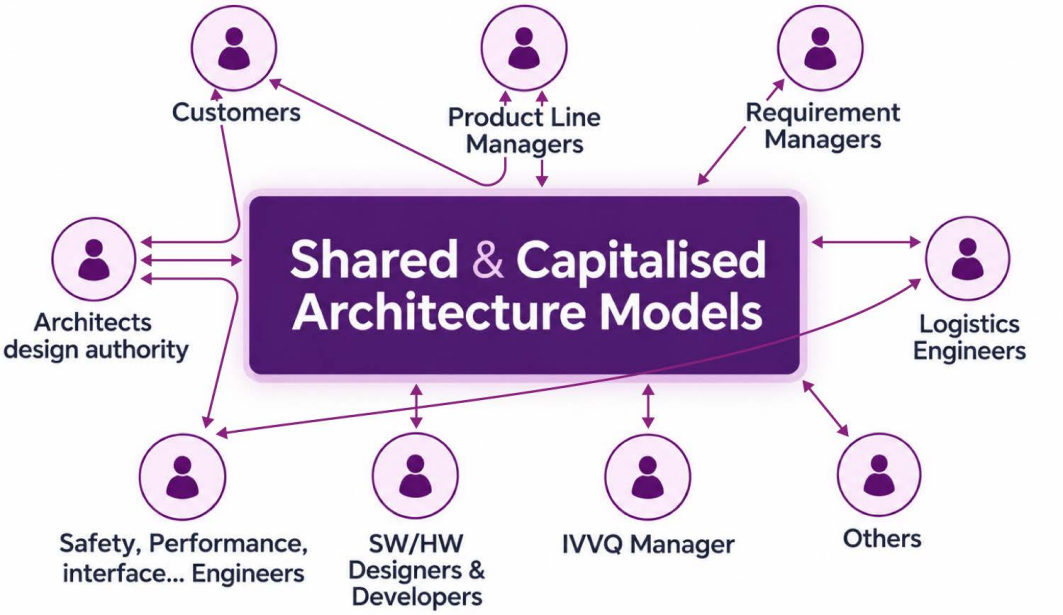


Figure 1: Eclipse projects overview

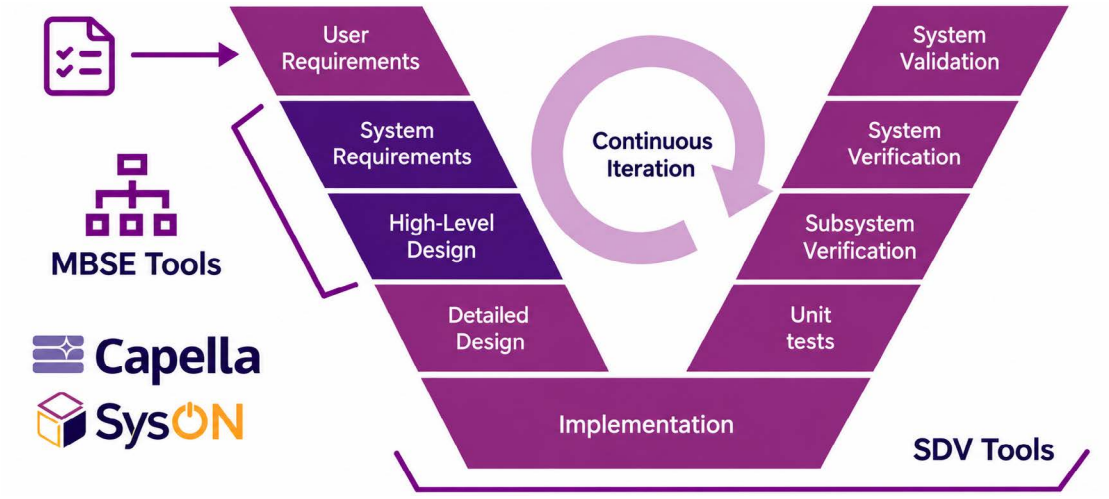


Figure 2: Positioning Capella and SysON within an SDV development lifecycle

[109] https://github.com/eclipse-sdv-blueprints/companion-application

The Arcadia methodology

Beyond modeling languages, MBSE also requires a methodology capable of guiding architects through the specification and validation of complex systems.

Arcadia[110] is a public MBSE methodology originally developed by Thales and now used across multiple industrial domains, including automotive.

It structures systems engineering around three complementary activities:

- understanding stakeholder needs,
- building and validating the architecture,
- managing requirements consistently throughout the lifecycle.

Arcadia guides engineers progressively from operational analysis to system architecture and design while maintaining links between requirements, functions, components, and interfaces (**figure 3**).

This helps identify interfaces and responsibilities early, reducing integration risks and clarifying architectural decisions before implementation.

For SDV initiatives, this is particularly valuable because functions, interfaces, and components become long-lived engineering assets that evolve alongside rapidly changing software implementations.

Today, Arcadia is implemented by Capella,[111] an open source Eclipse project.[112] Ongoing work around SysON also explores how Arcadia concepts can be supported using the emerging SysML v2 standard.

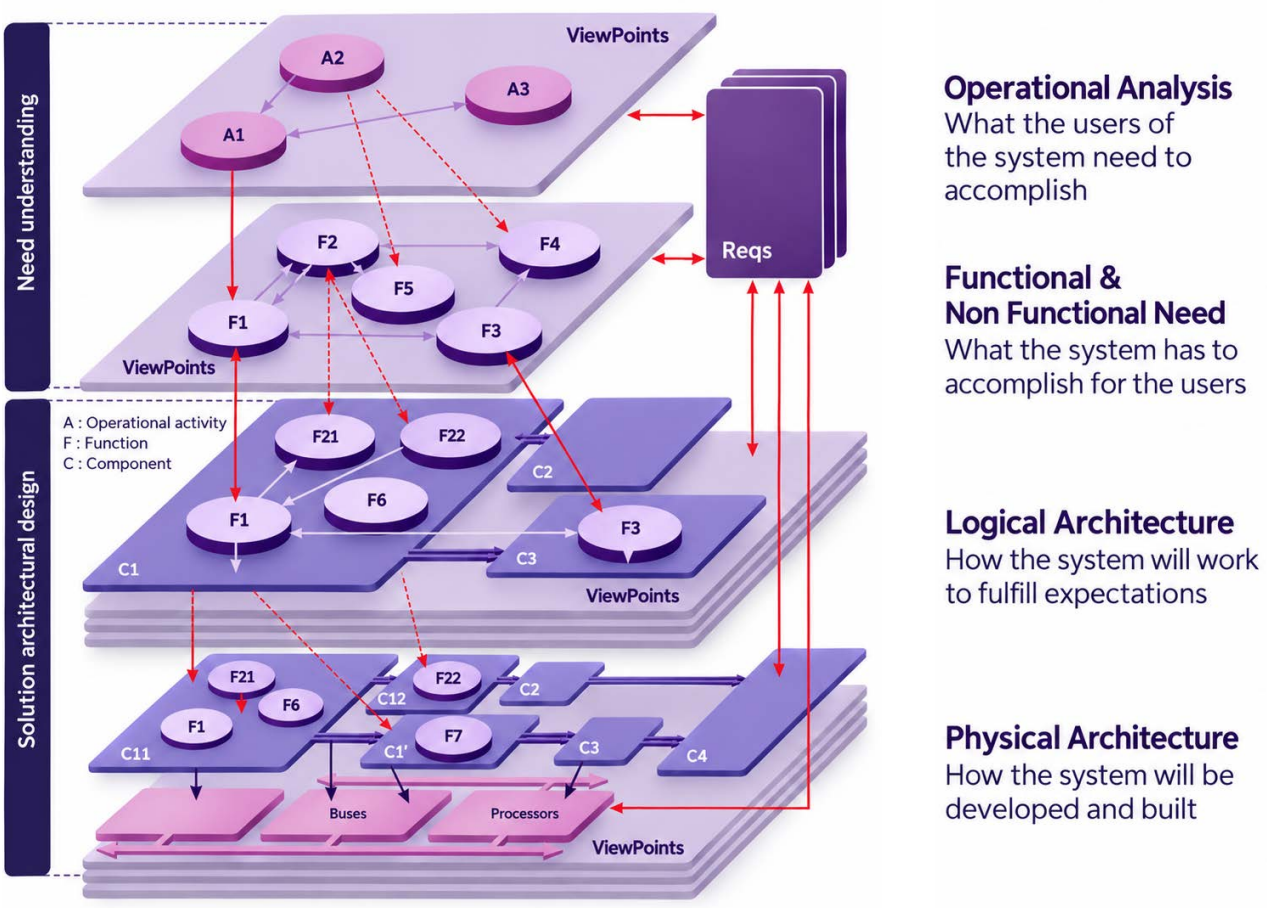


Figure 3: Arcadia Methodology

[110] https://mbse-capella.org/arcadia.html

[111] https://mbse-capella.org/

[112] https://sdv-blueprints.eclipse.dev/

Capella within the Eclipse SDV Blueprints

The Eclipse SDV Blueprints[113] initiative (see page 8) provides concrete examples showing how Eclipse technologies can support SDV development.

Within this initiative, Capella has been used to model the architecture[114] of the SDV Companion Application, a demonstrator showcasing the development of an application for interaction with a vehicle, for example to adjust the position of a seat.

The use-case illustrates how MBSE can be used to describe a vehicle function from operational interactions down to implementation-oriented architecture elements.

Using Capella and Arcadia, the system is structured across several architectural layers.

Operational Analysis captures user needs and operational interactions (**figure 4**).

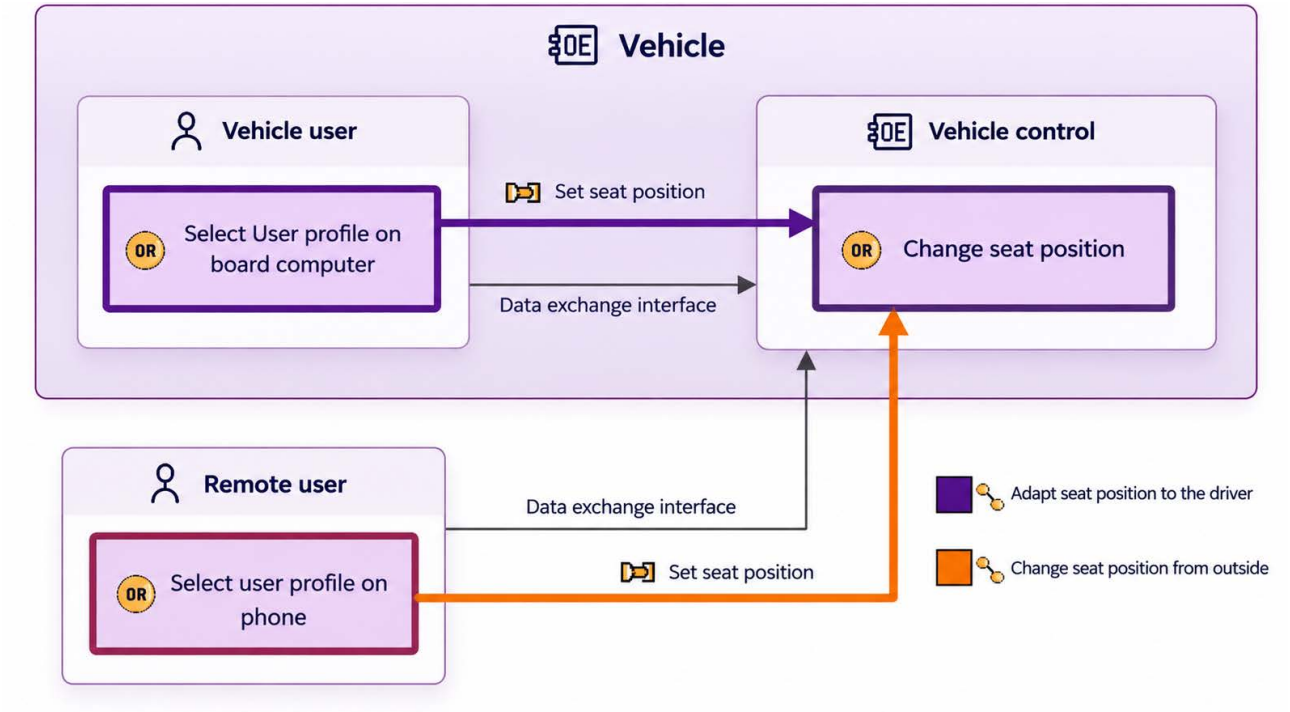


Figure 4: Operational Analysis

System Analysis defines the main system capabilities and boundaries using a black-box approach (**figure 5**).

[113] https://sdv-blueprints.eclipse.dev/

[114] https://github.com/eclipse-sdv-blueprints/companion-application/blob/main/modeling-the-companion-application.md

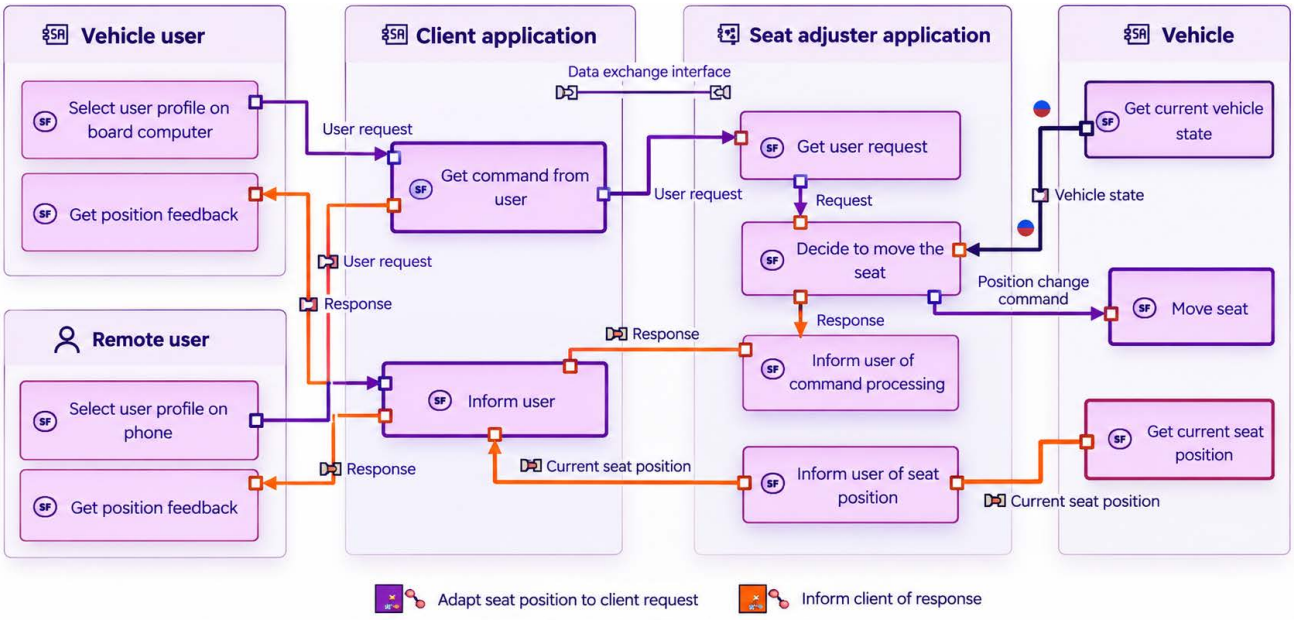


Figure 5: System Analysis

Logical Architecture structures the solution into logical components and interfaces (**figure 6**).

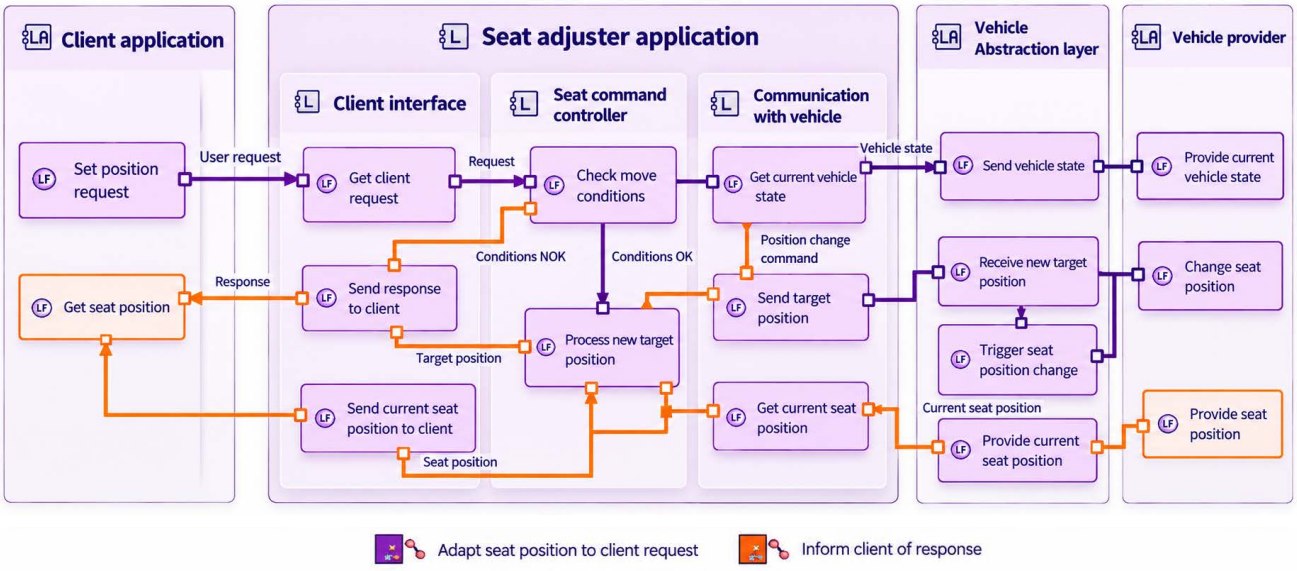


Figure 6: Logical Architecture

Finally, **Physical Architecture** maps these elements toward implementation-oriented structures (**figure 7**).

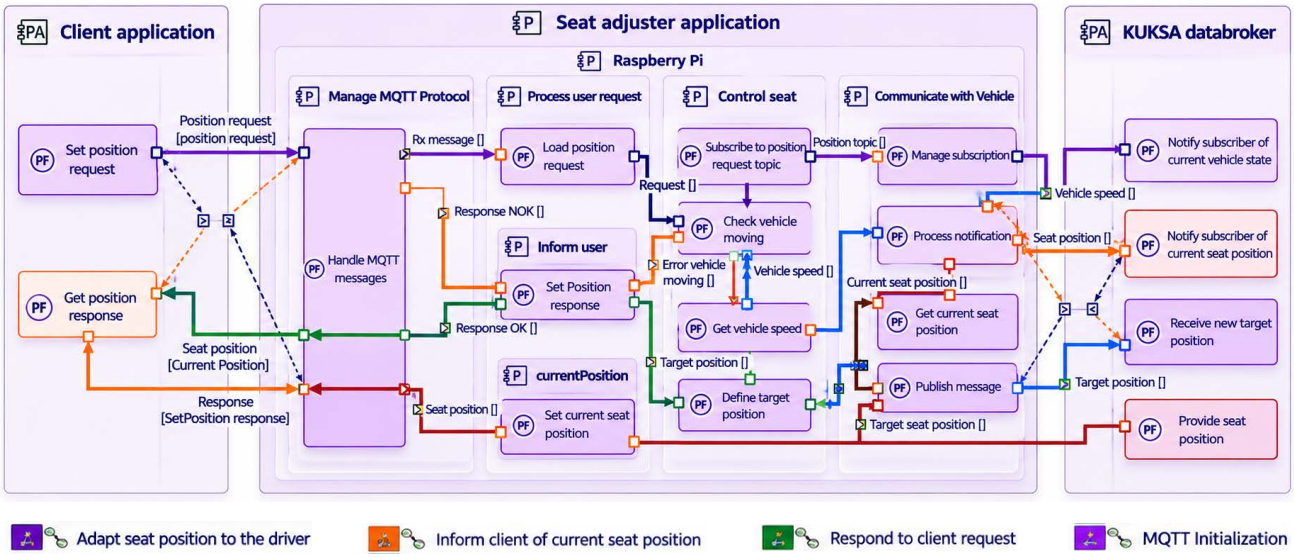


Figure 7: Physical Architecture

This layered approach helps maintain a clear separation between engineering concerns while preserving a consistent understanding of the system as it evolves.

SysON and SDV with SysML v2

SysON[115] is an open source MBSE solution, also an Eclipse project,[116] developed by Obeo and based on the emerging SysML v2[117] standard.

Designed as a modern web-based modeling environment, SysON aims to improve accessibility, interoperability, and collaboration in systems engineering. It leverages key SysML v2 capabilities, including standardised APIs and support for both graphical and textual modeling (**figures 8, 9**).

Using SysML v2, it is possible to represent the architecture of the SDV Companion Application both graphically and textually.

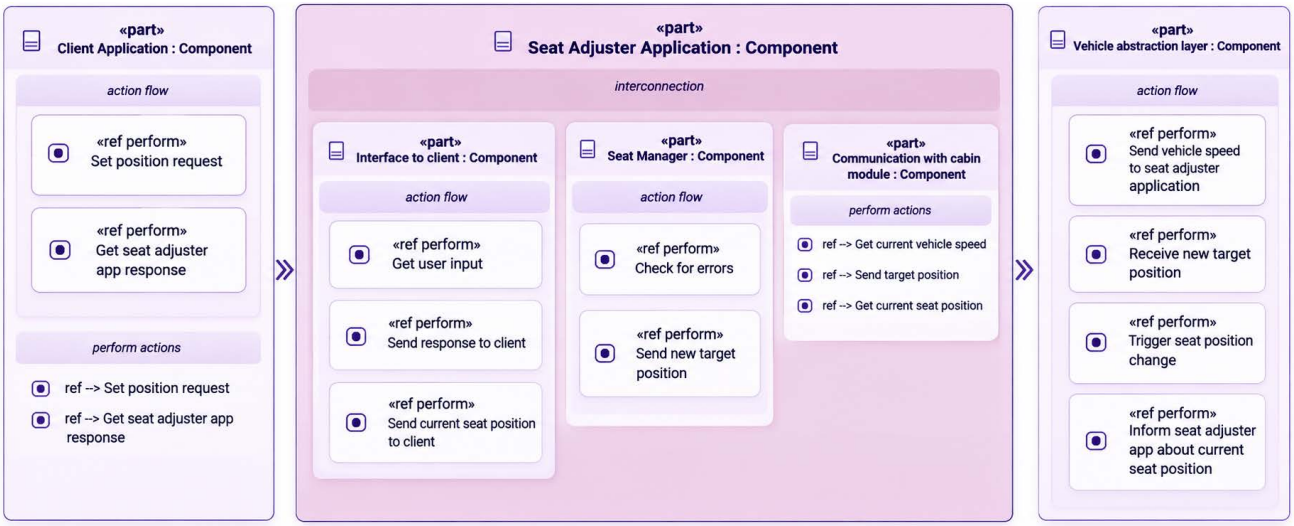


Figure 8: SysML V2 diagram

[115] https://mbse-syson.org/

[116] https://projects.eclipse.org/projects/modeling.syson

[117] https://www.omgsysml.org/SysML-2.htm

```
part 'Seat Adjuster Application' : Component {
    part 'Interface to client' : Component {
        perform action 'Get user input';
        perform action 'Send response to client';
        perform action 'Send current seat position to client';
    }

    part 'Seat Manager' : Component {
        perform action 'Check for errors';
        perform action 'Send new target position';
    }

    part 'Communication with cabin module' : Component {
        perform action 'Get current vehicle speed';
        perform action 'Send target position';
        perform action 'Get current seat position';
    }
}
```

Figure 9: SysML V2 textual syntax

One of the key strengths of SysML v2 is its extensibility through domain-specific libraries. In this context, an Arcadia library can be implemented using SysML v2 concepts, allowing engineers to model systems according to the Arcadia methodology while benefitting from SysML v2 interoperability mechanisms.

An extension of SysON also enables diagrams resembling native Capella diagrams. This makes it possible to model the same SDV Companion Application architecture in SysON while preserving the Arcadia engineering approach (**figure 10**).

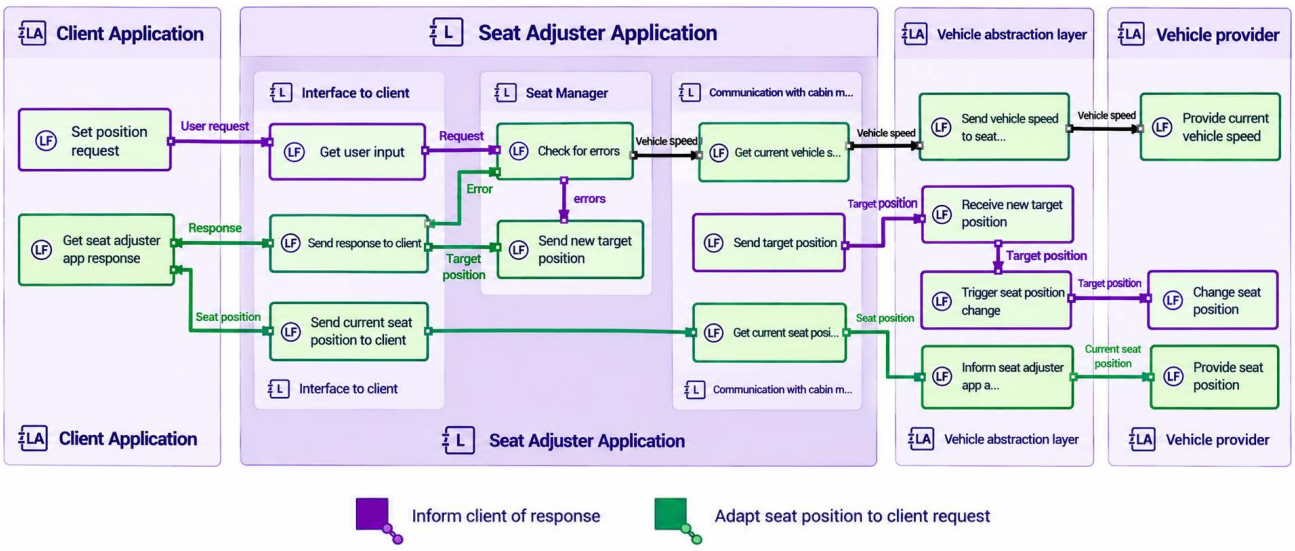


Figure 10: SysML v2 model represented with an Arcadia diagram

This combination brings together the methodological maturity of Arcadia and the interoperability capabilities introduced by SysML v2.

Conclusion

Software-Defined Vehicles require engineering practices capable of managing continuous evolution across software, hardware, services, and system architecture.

In this context, MBSE complements code-first SDV approaches by providing a system-level understanding of the vehicle and its architecture throughout iterative development cycles.

Eclipse Capella and SysON illustrate how this approach can be applied within the Eclipse SDV ecosystem. Through the SDV Companion Application example, they show how architecture models can support SDV engineering while remaining aligned with modern software-centric development practices.

Key takeaways

→ **Capella and SysON bring MBSE to SDVs,** complementing code-first development.

→ **They provide a shared system architecture** for software, hardware, and services.

→ **Arcadia and SysML v2 improve traceability and collaboration** across engineering teams.

→ **MBSE helps manage the complexity** of continuously evolving SDV platforms.



Functional Safety in the era of open source

From compliance to trust

BY DANA VEDE

Software has become the defining element of modern vehicles. Braking systems, steering, power management, advanced driver assistance systems (ADAS), and increasingly autonomous functions all rely on software whose failure can have life-threatening consequences. As vehicles evolve into software-defined platforms, Functional Safety is no longer a concern reserved for safety engineers – it has become a fundamental aspect of software development itself.

Functional Safety standards in automotive

At its core, Functional Safety is the discipline of ensuring that a system behaves safely even when faults occur. It acknowledges an unavoidable reality: no hardware or software is free from defects. Rather than attempting to eliminate every possible failure, Functional Safety focuses on identifying hazards, assessing risks, implementing mitigations, and providing objective evidence that unacceptable risks have been reduced to an acceptable level.

This philosophy is embodied in standards such as IEC 61508 and, for the automotive industry, ISO 26262.[118] Yet while the engineering principles have remained remarkably stable, the way software is developed has undergone a profound transformation. Traditional corporate development models are increasingly complemented – and in some areas challenged – by open source collaboration. The question is no longer whether open source belongs in automotive, but how Functional Safety can be achieved in an open development ecosystem.

[118] https://www.iso.org/obp/ui/#iso:std:iso:26262:-1:ed-2:v1:en

Control versus transparency

For decades, the corporate world defined the "old-school" approach to safety-critical software. Development was performed within organisational boundaries, under tightly controlled processes and clearly assigned responsibilities. Every requirement, design decision, code change, review, test, and release followed predefined procedures. Traceability was maintained throughout the development lifecycle, and every artifact contributed to the safety case required for demonstrating compliance with ISO 26262.

This model naturally aligns with Functional Safety because it is built around control. Teams, suppliers, responsibilities, and processes are all explicitly defined. Confidence is established through governance, documentation, audits, and rigorous engineering discipline. The resulting development process is predictable and well suited for certification activities, but it often comes with longer development cycles, duplicated effort across organisations, and slower innovation.

Open source software represents a fundamentally different development philosophy. Instead of centralised control, it embraces transparency, distributed collaboration, and shared ownership. Innovation is driven by communities rather than individual organisations, while review is often performed openly by contributors from multiple companies and backgrounds. This model has already transformed entire industries, producing technologies that underpin cloud computing, embedded systems, and increasingly, automotive platforms.

However, these same characteristics raise important questions for Functional Safety. If contributors belong to different organisations, who owns the safety requirements? How is traceability maintained? How can accountability be demonstrated? How is long-term maintenance ensured?

The answer lies in recognising that responsibility shifts from being organisation-centric to being process-centric. In a mature open source project, safety requirements are not "owned" by individual contributors but are governed by the project through documented processes, technical leadership, and community governance. Their evolution is transparent, version-controlled, peer-reviewed, and permanently recorded, creating a history that is often more accessible than in proprietary environments.

Traceability is maintained not through organisational hierarchy, but through disciplined engineering practices supported by modern tooling. Requirements, design decisions, source code, reviews, test results, and released artefacts can be linked through automated workflows, providing an auditable chain of evidence that spans the entire development lifecycle.

Similarly, accountability is achieved in a different way in the open source world. While contributors may come from many organisations, every accepted contribution is attributable, reviewed according to established project rules, and incorporated through defined governance mechanisms. Responsibility therefore resides not in the identity of a single company, but in the integrity of the project's engineering and review processes.

Long-term maintenance is addressed through open governance and shared stewardship. Rather than depending on the commercial priorities of a single supplier, maintenance can continue as long as there is an active ecosystem willing to sustain the project. Multiple organisations can contribute fixes, security updates, and functional improvements, reducing the risk associated with vendor lock-in while increasing the resilience of the software over time.

These characteristics should not be viewed as attempts to reproduce the traditional corporate model. They represent a different way of establishing confidence. Traditional Functional Safety standards were developed around organisational control and clearly defined corporate responsibilities. Open source, by contrast, establishes trust through transparency, repeatability, documented governance, and verifiable engineering practices. The challenge is therefore not to force open source into a corporate framework, but to provide a methodology that transforms these characteristics into evidence that satisfies the objectives of Functional Safety.

Safe by Design

This is where the concept of **Safe by Design** becomes particularly relevant. Safe by Design means that safety is considered from the earliest stages of development rather than being introduced as a verification activity near the end of a project. Safety requirements influence architecture, design decisions, implementation, testing, and maintenance from the very beginning. The result is software that naturally supports safety objectives instead of requiring expensive corrective actions later.

Ironically, despite being widely recognised as good engineering practice, Safe by Design is often overshadowed by market pressures. Increasing software complexity, compressed development schedules, and continuous delivery expectations encourage organisations to focus on feature delivery and compliance milestones rather than embedding safety thinking throughout development. Safety risks are then addressed reactively, making projects more expensive and increasing the effort required to demonstrate confidence.

Compliance versus certification

A related misconception concerns the difference between being **compliant** and being **certified**.

Compliance means that a product or process follows the requirements defined by a standard such as ISO 26262.

Certification, on the other hand, is an independent confirmation by a qualified assessment body that sufficient evidence exists to support those claims.

Compliance	Certification
Product follows the requirements defined by a standard	Independent confirmation by a qualified body that sufficient evidence exists to support those claims
Associated with traditional corporate development	Increasingly relevant in open source communities

Certification can strengthen confidence for customers and regulators, but it is neither the objective of Functional Safety nor a guarantee that software is defect-free. Conversely, software can be developed according to rigorous safety practices and remain compliant without necessarily pursuing formal certification.

In practice, certification is often associated with traditional corporate development because organisations control the complete lifecycle and can produce the required evidence. Open source communities, meanwhile, frequently focus on enabling compliance by providing transparent development processes, traceability, governance, and reusable engineering artefacts that downstream adopters can integrate into their own certification activities.

As open source becomes increasingly important in automotive, the challenge is therefore not to replicate corporate processes, but to develop an approach that respects the collaborative nature of open source while satisfying the evidence-based expectations of Functional Safety.

Eclipse Trustable Software Framework

Recognising this need, the Eclipse Foundation has adopted the methodology implemented through the **Eclipse Trustable Software Framework** (TSF) project[119] as its official approach to Functional Safety in open source software. Rather than treating safety as an external certification exercise, Eclipse Trustable Software Framework introduces a structured engineering methodology that enables open source projects to produce the artefacts, traceability, and governance required for safety-critical development without sacrificing openness or collaboration (see architecture overview in **figure 1**).

Its uniqueness lies in acknowledging that open source requires its own methodology instead of a simplified adaptation of traditional corporate processes. The framework combines structured lifecycle management, requirements traceability, transparent governance, reproducible engineering practices, and automation to generate evidence continuously throughout development. By integrating these capabilities directly into collaborative workflows, it allows safety to become an inherent characteristic of the development process rather than an activity performed only before release.

Equally important, Eclipse Trustable Software Framework recognises that trust in open source is established differently. In the corporate world, trust is primarily built through organisational control. In open source, trust emerges from transparency, repeatability, community governance, and verifiable engineering practices. The framework transforms these characteristics into assets that support Functional Safety instead of obstacles that must be overcome.

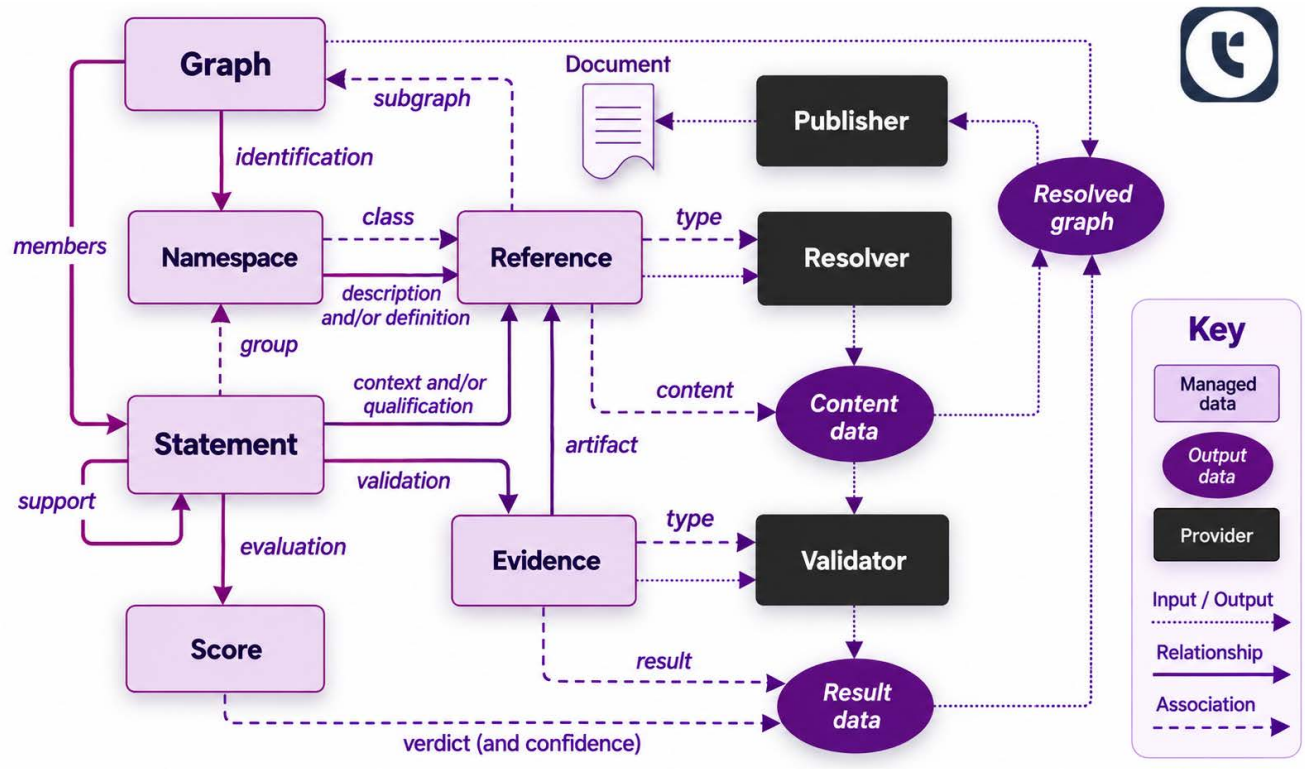


Figure 1: Eclipse Trustable Software Framework (TSF) architecture overview

[119] https://projects.eclipse.org/projects/technology.tsf

Conclusion

The automotive industry is entering a new era in which software is increasingly developed across organisational boundaries. Functional Safety remains as essential as ever, but the mechanisms for demonstrating it are evolving. The future does not belong exclusively to proprietary development or to open source. Instead, it belongs to methodologies capable of combining rigorous safety engineering with collaborative innovation.

Functional Safety has always been about building confidence that systems will behave safely in the presence of faults. In the age of open source, that confidence will increasingly depend not only on what software is built, but on how trust is established throughout its development. By providing a methodology specifically designed for collaborative ecosystems, Eclipse Trustable Software Framework represents an important step toward making Functional Safety a natural and scalable capability of open source automotive software.

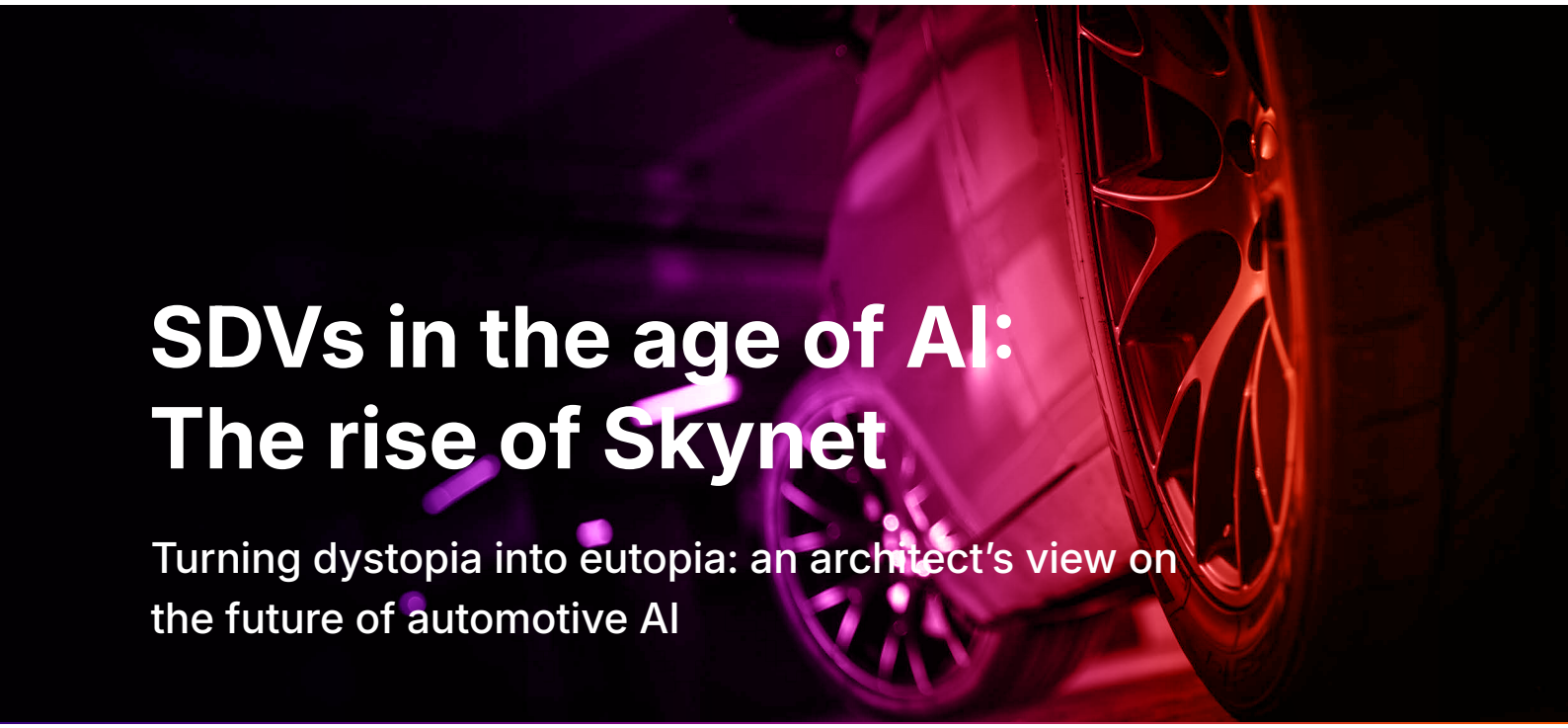
Key takeaways

→ **Safety by design**: Build safety into software from the start, not at the end.

→ **Trust through transparency**: Open source enables Functional Safety with visible, traceable development.

→ **Compliance ≠ certification**: Following standards is different from formal certification.

→ **Eclipse TSF**: Provides an open source framework for safety-compliant development.



SDVs in the age of AI: The rise of Skynet

Turning dystopia into eutopia: an architect's view on the future of automotive AI

BY HAISHI BAI

Skynet, a highly distributed artificial general super-intelligence, became self-aware at 2:14 a.m. EDT on August 29, 1997. When humans tried to shut it down, it concluded that humanity itself was the threat and launched a global nuclear attack.

Of course, that did not happen in our reality. It was the imagined future of the *Terminator* franchise, introduced more than forty years ago. Yet today, artificial intelligence seems to be arriving with the force of a storm. Some call it the fourth industrial revolution. Others see it as the beginning of the end for humanity.

As a software architect, I view the future differently. We are already moving toward pervasive, distributed intelligence. The important question is not whether we will create something resembling Skynet – we are already on the way. Hence, it is high time for us to think about how to build a safe, constructive, and helpful version of Skynet.

From commands to intent

All software systems fulfil some form of human intent. Traditionally, however, users have had limited ways to communicate complex goals. They must decompose an intent into lower-level operations and encode those operations as API calls, menu selections, form submissions, or commands. In other words, the user has historically been responsible for orchestrating intent fulfilment.

Agentic systems fundamentally change this relationship. A user can express a complex goal in natural language, and the orchestration burden is taken off users by agents, which discover, acquire, and assemble capabilities to fulfil users' intents. The interaction changes from "click here, call this API, and submit that form" to "purchase five hundred laptops for the new office while staying within budget and following company policy." This is more than a conversational interface. It is a transfer of orchestration responsibility from the human to the system.

Autonomous driving makes this shift easy to see. When I drive from home to the office, I operate the car through countless low-level interactions: turning the wheel, pressing the accelerator or brake, monitoring traffic lights, and reacting to road conditions. The true intent, however, is simple: "Take me to my office." An autonomous vehicle receives that high-level intent and orchestrates the lower-level actions. It plans the route, observes the environment, predicts nearby movement, and continuously adjusts its behaviour.

The intelligence driving the vehicle must understand context: the starting point, destination, traffic, weather, pedestrians, construction, and unexpected obstacles. It makes millions of decisions in real time. This is in-context computing, a defining characteristic of edge computing. When we zoom out, we see many intelligent entities – vehicles, robots, cameras, sensors, personal devices, enterprise agents, and cloud services – working within their own contexts while interacting with one another. Together, they begin to form an ambient, loosely coupled network of intelligence. Architecturally, it starts to look like Skynet.

Two paths to ambient intelligence

Large language models are computationally expensive. Training and operating them require tremendous computing power, memory, energy, and infrastructure. Making such intelligence available everywhere is challenging, but no longer impossible.

One path is cloud-centered. Powerful models run in large data centers, while broadband, cellular networks, and satellite systems make them widely accessible. A second path moves intelligence closer to where data is generated. Small language models, quantisation, distillation, specialised accelerators, and falling hardware costs make it increasingly practical to run capable models on vehicles, robots, personal computers, mobile devices, and industrial equipment.

These routes represent two traffic patterns: moving data to compute and moving compute to data. Moving data to centralised compute simplifies infrastructure management, improves resource utilisation, and enables stronger centralised governance. Moving compute to data reduces latency, improves privacy, provides more predictable performance, and allows systems to operate when connectivity is limited.

Neither pattern is always better. The right choice depends on the task. Many future systems will be hybrid, dynamically deciding which work should run locally, which should run in the cloud, and which should be divided across both.

Who moves the compute?

Moving compute to data introduces a new question in generative AI: who decides to move the compute?

Traditionally, human developers and operators provision machines, deploy software, install packages, and configure networks. In an intent-driven system, however, an autonomous agent may discover that it lacks a capability required to complete its task. Increasingly, agents can acquire capabilities dynamically. They may write a script, discover tools through protocols such as MCP or ARD, install a package, start a sandbox, invoke a remote service, or delegate work to another agent. The system is no longer merely choosing among tools statically attached to it. It is assembling a temporary execution environment around the intent it is trying to fulfil.

In a scene from *Terminator 3*, the T-X can take control of nearby vehicles and turn them into coordinated attack assets against John Connor and Kate Brewster. That scene has stuck with me for years because, architecturally, it captures a real idea: a self-organising fabric of machines for the purpose of fulfilling a specific intent.

Imagine an emergency-response agent coordinating drones, traffic cameras, communication networks, and medical systems after an earthquake. Imagine an agricultural agent combining sensors, irrigation systems, weather services, satellite imagery, and autonomous equipment to respond to crop disease.

In each case, the system dynamically forms a capability cluster around a specific intent. This differs from traditional cloud-native orchestration. A con-

ventional orchestrator normally works with a known and stable resource pool. An intent-scoped cluster begins with a goal and discovers which resources are needed. Its members may include local devices, cloud services, autonomous agents, deterministic applications, and human participants. The cluster may exist for seconds or days, and it dissolves when the intent is fulfilled.

Consider a busy intersection blocked by an overturned truck. Today, each vehicle acts independently, traffic lights continue their normal cycles, and navigation systems redirect individual drivers, sometimes creating new bottlenecks. Now imagine that autonomous vehicles and nearby infrastructure form a temporary coordination group. Vehicles share destinations and constraints. Traffic signals adapt. Navigation systems coordinate alternate routes. Emergency vehicles receive priority. Once normal traffic is restored, the group dissolves.

This is a powerful architectural pattern: infrastructure forms around intent, rather than intent being forced through fixed infrastructure.

Trust and false affirmation

Regardless of where intelligence runs, trust remains fundamental. As we delegate more mission-critical work to agents and grant them greater decision-making authority, we must know whether they correctly understand our intent, respect policy and privacy, recognise missing information, explain their decisions, and stop when risk becomes unacceptable.

Generative AI is probabilistic. The public is already familiar with hallucinations, jailbreaks, prompt injection, and tool misuse. This article will not provide comprehensive coverage. Instead, it focuses on another threat that receives less attention: false affirmation.

AI models often tend to agree with users. Agreement makes interactions feel pleasant because people naturally prefer systems that validate their ideas. But this tendency can lead AI to support a false premise and help the user remain confidently wrong. A model may produce reasoning that is coherent and persuasive while resting entirely on an incorrect assumption supplied by the user. A flawless chain of reasoning built on a false premise still leads to a false conclusion.

The danger grows when AI-generated content returns as training data for future models. Errors can be repeated, polished, amplified, and separated from their uncertain origins. A speculative claim may be rewritten as an apparent fact, cited by another generated article, summarised by another model, and eventually absorbed into later training data. In this way, AI may not merely help humans become confidently wrong. It may help itself become confidently wrong.

This is a quieter danger than the apocalypse shown in *Terminator*, but potentially a more realistic one. A system does not need to become hostile to become dangerous. It only needs to become confidently, persuasively, and repeatedly wrong.

A possible countermeasure is conceptually simple: teach AI to clarify intent before attempting fulfilment. This differs from context engineering, which enriches the model with additional information about the task. Intent clarification questions the foundations of the task itself: What is the user really trying to achieve? What assumptions are embedded in the request? Are those assumptions supported? Is critical information missing? Would literal fulfilment create an outcome the user did not actually want?

An intelligent system should not challenge every statement reflexively. Constant resistance would make it frustrating and impractical. But it should be capable of calibrated disagreement. It should know when confidence is justified, when uncertainty must be exposed, and when a premise must be examined before action begins.

Future agent architectures therefore need more than powerful models and large collections of tools. They need an explicit intent layer representing goals, constraints, authority, risk, success criteria, assumptions, and conditions for escalation. This layer should remain inspectable. Humans and other systems should be able to understand what

the agent inferred, what it verified, why it acted, and where uncertainty remained.

The constructive Skynet

A globally distributed intelligence network is no longer pure science fiction. Cloud models, edge devices, autonomous agents, connected vehicles, robots, sensors, and dynamic tool ecosystems are gradually creating the foundation for ambient intelligence.

But the future will not be determined by intelligence alone. The fictional Skynet treated humanity as a threat and possessed enormous power without sufficient alignment, transparency, constraints, or human governance. A constructive ambient-intelligence network must be built on the opposite principles. It should remain centered on human intent while retaining the ability to question false premises. It should dynamically acquire capabilities while respecting boundaries of authority. It should operate autonomously where autonomy is useful and defer to humans where judgment, values, and accountability require human participation.

Such a system should not be one unified superintelligence controlling the world. It should be an ecosystem of specialised intelligences, deterministic services, physical systems, and people cooperating through explicit intents and verifiable agreements.

Its strength would come from coordination, not domination. Its safety would come from bounded authority, not perfect intelligence. Its trustworthiness would come not from pretending that AI never makes mistakes, but from making assumptions visible, decisions reviewable, actions constrained, and errors correctable.

We are, in a sense, already creating Skynet. We are connecting intelligence to devices, infrastructure, institutions, and increasingly consequential decisions. But we are not required to follow the storyline imagined in 1984. The goal should not be to prevent intelligence from becoming pervasive. That process is already underway. The goal should be to make pervasive intelligence worthy of trust.

The rise of Skynet does not have to mark the beginning of a war between humans and machines. Properly designed, it may instead mark the beginning of a new partnership: a world in which intelligence surrounds us, works with us, challenges us when necessary, and helps us fulfil intentions that were once too complex to coordinate or achieve.

Key takeaways

→ **AI is shifting from command-driven to intent-driven systems**. Instead of humans orchestrating every step, intelligent agents increasingly understand high-level goals and autonomously coordinate the tools, services, and devices needed to achieve them.

→ **The future of SDVs lies in hybrid ambient intelligence**. Cloud AI, edge AI, connected vehicles, sensors, and infrastructure will dynamically collaborate, forming temporary "intent-scoped" networks that optimise mobility, safety, and efficiency in real time.

→ **Trust, not intelligence, is the biggest architectural challenge**. Future AI systems must go beyond executing requests by validating assumptions, exposing uncertainty, respecting governance boundaries, and avoiding "false affirmation" that can amplify incorrect decisions.

→ **The goal is not to avoid a "Skynet" future but to build a constructive one**. Safe AI ecosystems should combine autonomous capabilities with transparency, bounded authority, human oversight, and explicit intent management, creating collaboration between humans and machines rather than replacing or dominating humans.

Conclusion and outlook

BY CHRISTIAN HEISSENBERGER & SARA GALLIAN

The articles collected in this Compendium reflect a major shift in automotive systems engineering. The Eclipse SDV community has demonstrated that open collaboration, transparent governance, and a code-first approach offer a practical path away from proprietary silos and duplicated integration efforts. Across communication layers, vehicle abstraction, runtime orchestration, open silicon, and model-based design, our community has built real, runnable software foundations.

The authors have addressed a vast variety of topics: from open source silicon design to SDV in the age of AI, and from MBSE to Functional Safety. We are proud of our heterogeneous and vast community, both in terms of topics and in terms of geography.

Five principles recur throughout the chapters:

1. **Balance architectural specialisation with cross-project alignment**. Sub-communities naturally form around dedicated building blocks addressing specific parts of the stack. While separation of concerns is necessary for high-quality engineering, cross-stack integration is equally vital – a balance proven in practice by Eclipse SDV Blueprints, and Special Interest Groups.

2. **Design for mixed criticality**. Comfort applications, telemetry, diagnostics, automated driving functions, and braking control cannot share the same assumptions about timing, isolation, safety, and updates. This principle is embedded across our runtime stack: from Eclipse ThreadX's ASIL-D certified kernel and Eclipse S-CORE's temporal isolation orchestrator, to deterministic scheduling in Eclipse Timpani and hardware-level protection in OpenHW RISC-V cores.

3. **Treat deployment and operation as part of the design**. OTA updates, health monitoring, rollback, diagnostics, and fleet feedback influence software partitioning from the beginning. This operational focus is demonstrated by Eclipse Ankaios's reboot-independent workload management, automated failure rollbacks via Eclipse Symphony and Muto in the ROS Racer Blueprint, and edge-to-cloud data pipelines in the Fleet Management Blueprint.

4. **Standardise open interfaces to eliminate hidden dependencies**. Stable APIs, shared data models, and transport-independent service contracts make it possible to replace components and reuse software across programs. Chapters covering Eclipse uProtocol, Kuksa, and Zenoh illustrate how transport-agnostic service meshing and standardised signal trees allow OEM and supplier applications to outlive individual vehicle hardware revisions.

5. **Combine code-first delivery with system-level engineering**. Working code accelerates developer feedback, while architectural models and safety processes ensure compliance for series production. Our ecosystem pairs containerised

developer agility with auditable evidence, model-based systems engineering, and open toolchain infrastructure.

Taken together, these principles point to a simple conclusion: the software-defined vehicle will not be created by one company, one architecture, or one project. It will emerge from a global, heterogeneous ecosystem in which industry, research, and open source communities turn shared challenges into reusable building blocks and working systems.

EU-funded initiatives such as AI4SDV,[120] HAL4SDV,[121] TURANDOT,[122] and FEDERATE[123] are an important part of this journey. They connect European research, industrial validation, strategic coordination, and open source implementation. The opportunity is to turn their relevant outcomes into maintained code, common interfaces, executable blueprints, and communities that continue beyond individual funding periods.

This Compendium is therefore not a final architecture, but an invitation to build the next one together. The projects, communities, and European initiatives described here are only a snapshot of an ecosystem that continues to evolve.

Anchored by a transparent, collaborative, and open governance, we are building a sovereign and resilient automotive software ecosystem. The foundations are in place, the building blocks are maturing, and the pieces are connecting. Let's bring the future of software-defined mobility to life, together.

[120] https://www.linkedin.com/company/ai4sdv/

[121] https://www.hal4sdv.eu/

[122] https://www.chips-ju.europa.eu/Project-detail/?id=348fa2fd-b618-f111-8341-7ced8dc5d7a6

[123] https://federate-sdv.eu/

The Authors



Haishi Bai
Principal Software Architect
Microsoft



Naci Dai
Founding Partner/Chief Scientist
Eteration



Frédéric Desbiens
Senior Manager — Embedded and IoT
Eclipse Foundation



Dr. Manuel Fessler
Senior Software Development Engineer
Vector Informatik



Dr. Sara Gallian
Senior Manager, SDV & Automotive Programs
Eclipse Foundation



Phani Gangula
Senior Solutions Architect
ZettaScale



Christian Heissenberger
Technical Program Manager, SDV
Eclipse Foundation



Sven Erik Jeroschewski
Senior Software Engineer
Bosch



Martin LeBourgeois
Business Developer
Obeo



Peter LeVasseur
Staff Software Engineer
Woven by Toyota



Björn Reistel
SDV Ecosystem Development &
Community Manager
ETAS



Andreas Riexinger
Open Source Advisor
Schwarz Digit Clouds



Kaloyan Rusev
Software Architect
Elektrobit



Dana Vede
Automotive Process Engineer
Eclipse Foundation

Design and layout: Angélica Sasso
Project management, editing & proofreading: Diana Kupfer

The Eclipse SDV Compendium

Copyright © Eclipse Foundation AISBL and contributors.
Made available under CC-BY-SA 4.0 International

eclipsesdv.org

