

veecle Operating System (OS)

Technical White Paper

veecle OS is a runtime and programming model for machines made of many small computers (ECUs, domain controllers, HPCs). It helps you build software that stays understandable as the system grows in size, hardware diversity, and communication complexity.

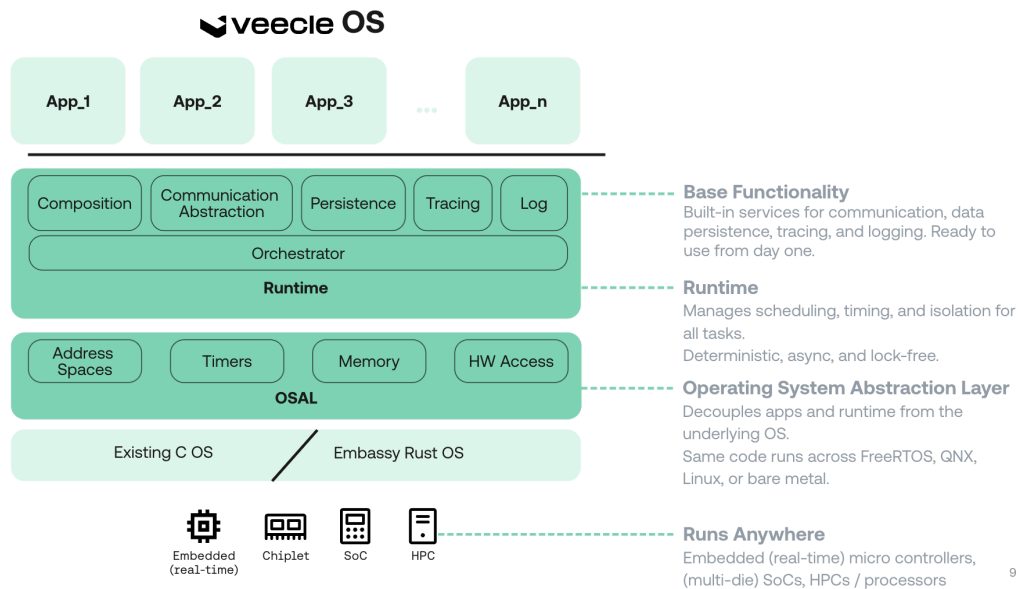
Instead of writing one big control loop with lots of shared state, you split functionality into small, independent async components (“Actors”). Actors exchange data through typed data channels (“Slots”) and react when new data arrives.

Mentally, it works like this: sensors publish data → actors transform/decide → actuators consume results. The runtime provides the glue (data-flow, scheduling, and observability) so you don’t end up hand-writing fragile plumbing code.

It provides:

- A consistent programming model across heterogeneous targets via an OS abstraction layer (so the same app logic can run on different OSes and CPUs).
- A typed, versioned data store with reader/writer semantics that makes data dependencies explicit and debuggable.
- Integration points for cross-process and cross-device communication, including code-generated adapters for common vehicle protocols (where applicable).
- Built-in telemetry hooks so you can understand timing, causality, and data flow in a running system.
- A workflow that fits testing and CI from day one (examples and tooling live alongside the runtime).
- Automatic testing, telemetry collection, and CI integration.

High-Level Concepts



Core Idea

- Customers develop and deploy **Applications**, logical bundles of Services belonging to a common business logic.
- Applications are composed by **Services**, logical working units that aggregate Actors. Services define ASIL levels, versioning, as well as define APIs and requirements.
- Services are decomposed into **small, asynchronous units** executed by the Veecele runtime, called Actors - these are what customers code or import.
- Actors can be templated to allow reuse as well as shared between libraries and services.
- Actors behavior is defined by **data-flow**, not control-flow: Actors are not concerned with hardware, middleware or operating system details.
- Communication between actors is managed by the Veecele Runtime and is **strongly typed, validated, middleware-agnostic, and optimised**.

Key Terms

Term	Meaning	Scope
Actor	Asynchronous unit of work reacting to data availability. This is the only entity that has to be coded by application developers.	Execution-level primitive.
Service	A logical grouping of Actors that together implement a function. This is configured and maps to existing templates within the codebase.	Feature-level component.

App	A bundle of Services forming a product-level capability (e.g. "Lighting App"). They can run on different targets, hardware and OSes.	System-level capability.
Runtime	Schedules and executes Actors and handles messaging, timeouts, isolation. No Operating System dependencies - fully no-std.	Veecle OS Core Library
Orchestrator	Supervisory component responsible for coordinating execution, memory layout, and communication between Runtimes.	Veecle OS Core Library
Store	Veecle's distributed asynchronous database that manages channels and ensures async data access is correctly mapped to actors.	Veecle OS Core Library
Operating System Abstraction Layer	Provides unified API for scheduling, timing, and memory behavior across OSes. Can be extended to support more features or systems.	Veecle OS Core Library
no-std	Rust core bare-metal library - no Operating System or Heap allocation support. Used to minimise dependencies and requirements.	Rust Core Library
Meta-Model	Unified model containing all required configuration to compile and validate Veecle OS. Enables automated CI/CD.	Architecture source of truth.

The Runtime

Veecle OS executes application logic using an async-first runtime. Instead of creating an OS thread per component, the runtime drives many independent components ("Actors") inside a single event loop by polling them when they are ready to make progress.

An Actor is written as an async function. It runs until it hits a yield point (an await), such as "wait for new input data", "wait for a timer", or "publish an output". At that moment it cooperatively yields control, allowing other ready Actors to run. This keeps execution predictable: switching between Actors happens only at explicit yield points, not at arbitrary instruction boundaries.

This is why "cooperative" is not the opposite of "real-time". In Veecle OS, you still build real-time systems the classic way: by controlling priorities, isolating critical work, and bounding latency. The difference is that inside a runtime instance, scheduling is explicit and engineering-driven (yield points), rather than hidden preemption that can occur anywhere.

Real-time integration options

There are two common deployment patterns, depending on how strict your timing and isolation needs are:

a) **RTOS-managed runtimes (classic priority model)**

Run multiple Veecle OS runtime instances as separate RTOS tasks/threads, each with its

own priority (and optionally its own core). The RTOS enforces hard preemption and priority between runtime instances, while each runtime instance cooperatively schedules its internal Actors. This is a pragmatic way to keep critical control loops isolated from less critical workloads without forcing every component into its own OS thread.

b) **RTOS-native async executors (interrupt-driven wakeups)**

On targets that provide an RTOS-native async executor (e.g., an Embassy-style model), Actors can be woken by interrupts and scheduled according to executor/RTOS semantics. Conceptually, Veecele OS still uses async Actors, but “who gets to run next” can be influenced by the underlying system’s real-time mechanisms, rather than only by cooperative fairness.

Multiple runtimes and orchestration

A single Veecele OS instance can contain one or more runtimes. Each runtime manages one or more Actors and is deliberately decoupled from the underlying platform. Coordination between runtimes (and communication across cores or ECUs) is handled at the orchestration layer. Platform-specific services such as time, timers, and OS integration are provided through the Operating System Abstraction Layer (OSAL).

Actors

Actors are small, deterministic, asynchronous handlers. This is where code is actually written by the application developer. Actors are fully managed by the Runtime and are triggered by:

- The arrival of new data (e.g., `vehicle_speed`).
- Timeouts, events or scheduled intervals.
- External events originating from hardware, sensors, or other services.

These handlers never block; instead, they yield control back to the runtime whenever they need to wait. This design enables extremely high CPU utilization (often approaching 100%) without incurring traditional context-switching overhead.

Services

Services are bundles of actors belonging to the same context. They can be used to bundle a group of actors into a single cohesive import to simplify code reuse and maximise usability. Most policies, configurations, contracts, etc. are defined at Service-level, not Actor-level.

Services **are not implemented**, they are configured via our meta-model (see sections below). Therefore, services:

- Are Organizational structure grouping related Actors.
- They are configured via our modelling framework.
- Defines clear input/output data of the entire actor bundle.
- Hold configuration, state, timing, access rules, middleware mapping

Applications

Applications are “packages” that encapsulate a complete feature. They serve as logical units that make it easy to organise functionality into structured modules. For example, “**Wiper Control**” could be an application. An App exposes APIs such as `struct Wiper { }` and defines rulesets specifying where the code is allowed to run, what requirements it has, what permissions it needs, and so on.

Applications are **hardware-agnostic**: they define the logical group of services that make up the application. However, individual services may require specific hardware capabilities. For example, the WiperController service might require (1) an ASIL-D microcontroller and (2) a specific SPI driver.

In general, each application corresponds to a single project and its associated meta-model file (model.toml), where all requirements, APIs, actors, and related definitions are described.

```
# Automotive Gateway Example
[application]
name = "gateway"
version = "2.1.0"
description = "Multi-ECU automotive gateway system with CAN and Ethernet"

# Protocol interfaces for external communication (defined inline)
[interfaces.protocol.can.SensorData]
can_id = 0x123
frame_format = "standard"
signals = [
  { name = "temperature", start_bit = 0, length = 16, scale = 0.1, offset = -40.0 },
  { name = "pressure", start_bit = 16, length = 16, scale = 0.01, offset = 0.0 }
]

# Protocol interface for SOME/IP (inline)
[interfaces.protocol.someip.VehicleService]
service_id = 0x1234
instance_id = 0x5678
methods = [
  { name = "GetSensorData", method_id = 0x01, input = [], output = ["VehicleData"] },
  { name = "GetDiagnostics", method_id = 0x02, input = [], output = ["DiagnosticMessage"] }
]

# === Service GatewayService === #
[services.GatewayService.actors.can_receiver]
```



```
path = "linux_can_interface::LinuxCanInterface"

[services.GatewayService.actors.protocol_converter]
path = "can_to_ethernet::CanToEthernet"

[services.GatewayService.actors.data_converter]
path = "data_converter::DataConverter"

[services.GatewayService.actors.someip_service]
path = "someip_server::SomeipServer"

# == Deployment == #
[deployment.gateway_ecu]
services = ["GatewayService", "veecle::TelemetryService"]
platform = { os = "linux", arch = "x86-64" }
```

Operating System Abstraction Layer

Veecle OS isolates all “things that depend on an operating system” behind a small, explicit abstraction layer. Application code (Actors) talks to OSAL, not to FreeRTOS/QNX/Linux APIs directly. This keeps application logic portable and keeps the runtime model consistent across targets.

The OSAL’s goals are:

- Provide a unified API for time, networking, logging, and thread-related metadata, so Actors can rely on the same concepts everywhere.
- Make portability a normal case: the same Actor code can run unchanged across environments, while only the OSAL backend changes.
- Keep the abstraction surface intentionally small so that porting to a new platform is a bounded engineering task, not a rewrite.

What OSAL actually covers (today)

OSAL is not “a wrapper around everything an OS can do.” It focuses on the minimal set of capabilities Veecle OS needs to run and to be observable:

- **Time:** platform-agnostic Instant/Duration, sleeping, timeouts, and “what time is it?” through a TimeAbstraction trait.
- **Networking:** TCP/UDP traits designed to work on embedded as well as desktop targets (for example, OSAL’s TCP socket model is intentionally constrained to one active



connection per socket instance to avoid hidden complexity and to match embedded reality).

- **Logging:** a `LogTarget` trait that lets Veecele OS route log output consistently while using a platform-specific time source for timestamps.
- **Thread metadata:** a minimal `ThreadAbstraction` (e.g., stable thread IDs) mainly to correlate telemetry/logs in multi-threaded environments.

How it's structured

- The API lives in `veecle-osal-api` as a set of Rust traits and small supporting types (e.g., OSAL-owned `Instant/Duration` to avoid leaking platform types into portable code).
- Concrete implementations live in separate crates:
 - `veecle-osal-freertos` for FreeRTOS-based targets.
 - `veecle-osal-embassy` for `no_std` single-core systems using an Embassy-style async model (with an explicit limitation to single core/thread).
 - `veecle-osal-std` for “std environments” using Tokio and standard library facilities (this is typically how Linux and QNX-class POSIX targets are served).

Extending to new platforms

Supporting a new OS or hardware platform typically means implementing the existing OSAL traits for that environment. If a genuinely new capability is needed, OSAL grows by introducing a new trait with a narrowly scoped responsibility, rather than expanding existing traits into “god interfaces.” This keeps the portability boundary sharp: new platform work stays in the OSAL crate, and Actors remain platform-agnostic.

The Orchestrator

The **Orchestrator** is the supervisory component of Veecele OS and is responsible for coordinating execution, memory layout, and communication topology at both deployment time and runtime. It manages the Veecele Runtimes, isolates them into threads/tasks/sandboxes, ensures that IPCs



are correctly configured and optimized, and securely connects the Runtimes to the rest of the system.

It is imperative to point out that the orchestrator **does not implement** custom scheduling, memory or networking. Rather, it utilises the OSAL and Meta-Model to leverage the underlying Operating System capabilities to configure and manage the Veecele OS instance. In details:

1. Runtime creation, configuration and management

- Decides where each Runtime instance runs.
- Set priorities, configuration and manage the runtimes.
- Uses meta-model constraints: timing, communication, required capabilities.

2. Memory and Data Management

- Ensures correct visibility of data between runtimes.
- Validate IPC channels and networking medium. Ensure isolation when required.
- Configure and manage shared-memory where zero-copy is possible.

3. Telemetry and Health Management

- Collect health information from each Runtime and ensure the correct state.
- Collect and monitor telemetry information to ensure full visibility.
- Acts as a management point to dynamically configure the Runtimes.

4. Enforces Scheduling Constraints

- Ensures correct preemption rules are enforced.
- Ensures deadlines and timeouts are detected and handled.
- Monitor for jitters or unexpected delays.

5. Communication Routing and Management

- Selects the transport layer per communication edge based on model.
- Managed traffic, optimisation and ensures versioning.
- Maintain and establish dynamic connections.

The orchestrator is developed itself based on the Actor model and can therefore be extended, tested and deployed in the same way.

Telemetry and visibility

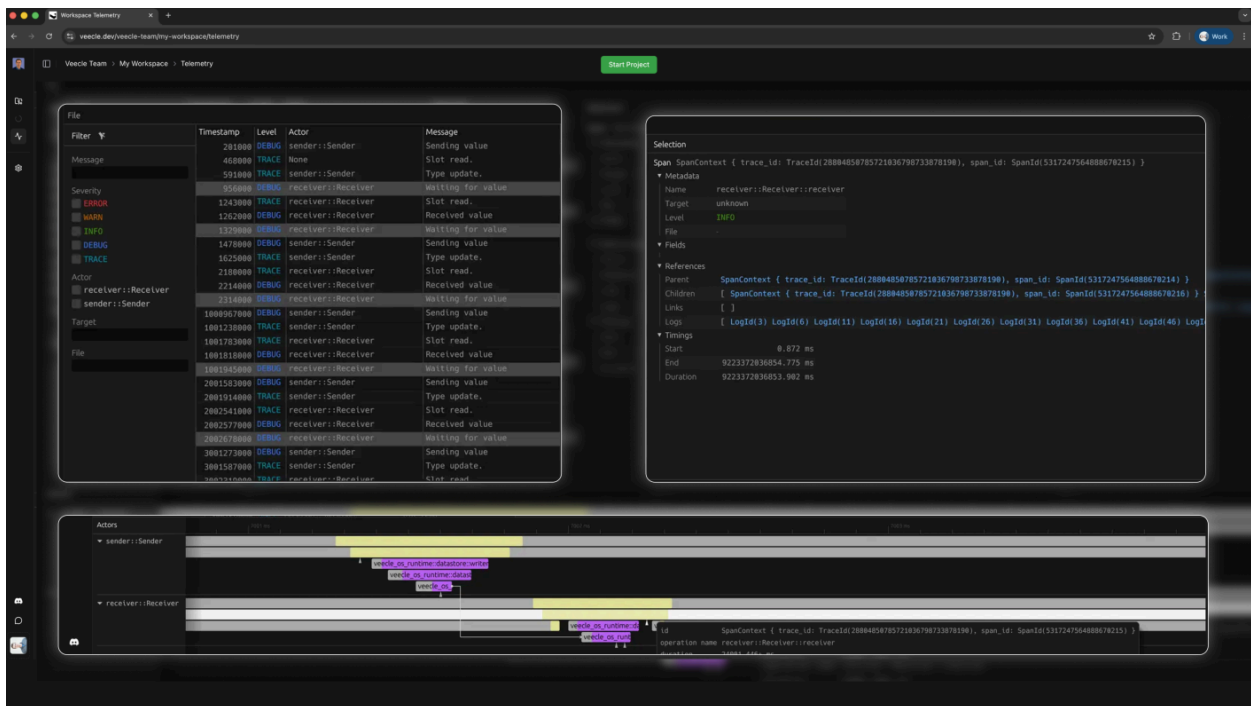
To maximise visibility and usage, Veecele integrates **Telemetry** across all its components. Fundamentally, every function in Veecele is (or can be) annotated with **#[telemetry]** to automatically collect telemetry information. Telemetry can be configured both at compile time and at runtime.



With telemetry enabled, every action generates a unique trace that is serialised and sent to a dedicated actor, which then pushes the data to a backend. Data is serialised using the OpenTelemetry format to ensure seamless integration with third-party tracing tools.

Telemetry enables end-to-end data-flow analysis and supports the inspection of asynchronous boundaries, function complexity, and resource usage. It supports any data type that VeeCle supports, and can be freely customized by customers. Likewise, standard log messages can be used – they are simply mapped into telemetry events.

An example of basic telemetry visualisation in our Studio is shown below. Many additional features – such as jumping to code, more human-readable logs, visual call graphs, and more – are currently under development.



Data-Flow and Store concepts

All data used in Veecele are **strongly typed Rust types**. By design, types must be safe to be moved across memory and threads. Almost all types of arbitrary complexity are supported.

Veecele provides procedural macro and tooling to import data from definition files such as **DBC, SOME/IP XML, DDS IDL, Android VHAL properties** and so on. Further, developers are always free to manually define or import data types. Traits can be used to extend the capabilities on the data.

The Store: Distributed Data Access

The **Store** is the core element managing data in Veecele and is never exposed to application developers, and is an implementation detail of Veecele OS. Each runtime has its own instance of the Store and access is therefore regulated by the runtime itself, and since each runtime works in single-threaded mode - no locks, nor complex memory models are required.

Actors are given, upon creation by the runtime, only **Read** or **Write** handles to the store. These handles are the only entity which allow Actors to interface with the store. Actors can:

- Use **Read** handles to await a specific update/event on one or more data conditions.
- Use **Read** handles to immediately access, without copy, the data within the Store.
- Use **Write** handles to push back to the store an optional output.

The Store can be used to model time and other operating system events. These can be exposed to actors as any other data type. The store, upon creation, is configured by the runtime to handle timeouts, default values, permissions, quality of service, and other behavioral parameters. These settings can be extended by the customer either by defining new modeling parameters or by extending the underlying storage abstraction to introduce custom logic and policies.

Cross-task (IPC) and cross-device (Networking) communication

The Store is designed to detect whether data is being awaited or generated within the same Runtime. If this is the case, no additional action is required – the Runtime simply moves memory references internally. However, if the data must cross Runtime boundaries, the Store uses direct inter-process communication (IPC) techniques (including zero-copy mechanisms) to transfer the data to the Orchestrator.

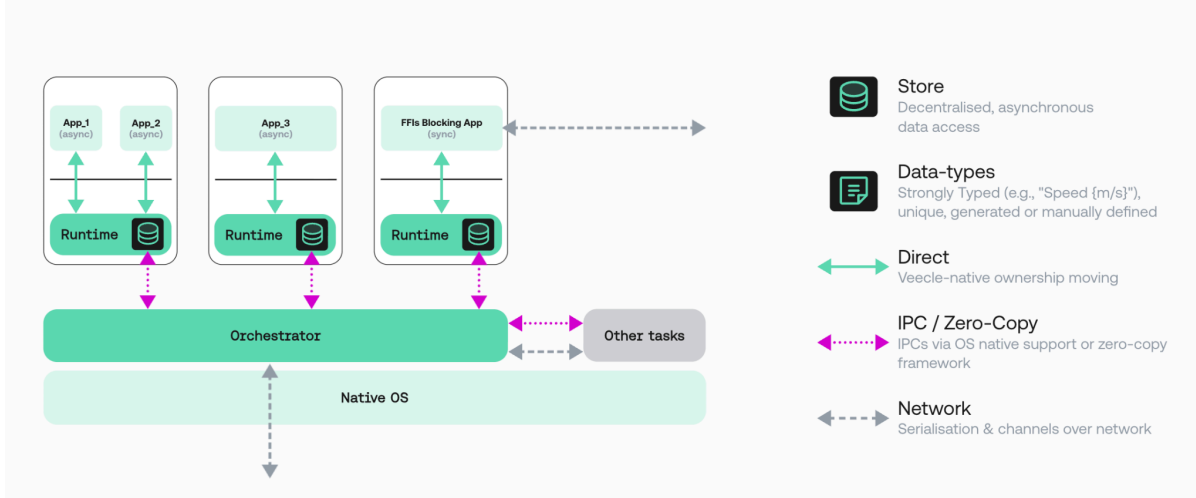
The Orchestrator then decides whether the data should be sent over a network interface (e.g., DDS or CAN) or forwarded to another Runtime. When forwarding to another Runtime, the system

can transfer memory directly, with or without copying, depending on the chosen isolation strategy.

For network transmission, it is important to note that the Orchestrator does not perform the transmission itself. Instead, it forwards the data to a dedicated, managed Runtime whose actors handle serialisation, encryption, transmission, and connection management. Veeacle provides built-in actors for these tasks, but they can be customized or completely replaced. These actors behave like any other Actor within the system.

Veeacle OS - Communication

All communication is abstracted — middleware, serialization, and ownership management are handled by the runtime.



During compilation, the Runtime and Orchestrators are generated with full knowledge of which IPC or network protocol should be used for each interaction. As a result, the runtime overhead is minimal: the Store is guaranteed at compile time to pre-allocate the required resources and already knows which data needs to be transferred to the Orchestrator (or vice versa), as well as when and how this transfer should occur.

This approach ensures that network or IPC design choices never leak into application-level actors. It also allows the underlying communication methodology to be changed at any time without affecting the Runtimes that execute application logic. Currently, Veeacle supports automated generation and integration for:

- CAN
- DDS
- SOME/IP
- Android VHAL

Creating an Application (Developer Workflow)

Veecle allows developers to write an application once and deploy it across heterogeneous hardware targets: Safety MCUs, HPC Linux processes, and Android IVI. This without rewriting any application logic. The entire process is driven by the Meta-Model, which defines data-flows, hardware targets, communication patterns, safety constraints, and timing requirements.

1. Code the Application Logic

The developer writes logic in a functional, data-driven, async style:

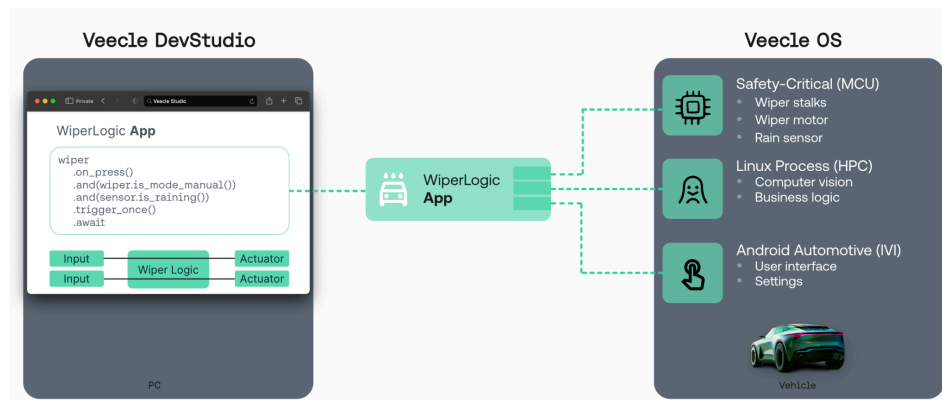
```

/// Simplified example of business logic of a Veecle Actor
#[actor(wiper_tear_drop_function)]
async fn handler(wiper: ReadWrite, rain_sensor: Read) {
    // Async Trigger condition
    wiper
        .on_press()
        .and(!wiper.is_mode_manual())
        .and(sensor.is_raining())
        .await;

    // Control logic upon awakening
    writer.trigger_once();
}

```

This defines only the logical behavior, not the hardware integration, timing, middleware, or any other implementation details. No communication mechanisms, data encodings, threads, or event loops are referenced at this stage.



2. Meta-Model as the Source of Truth

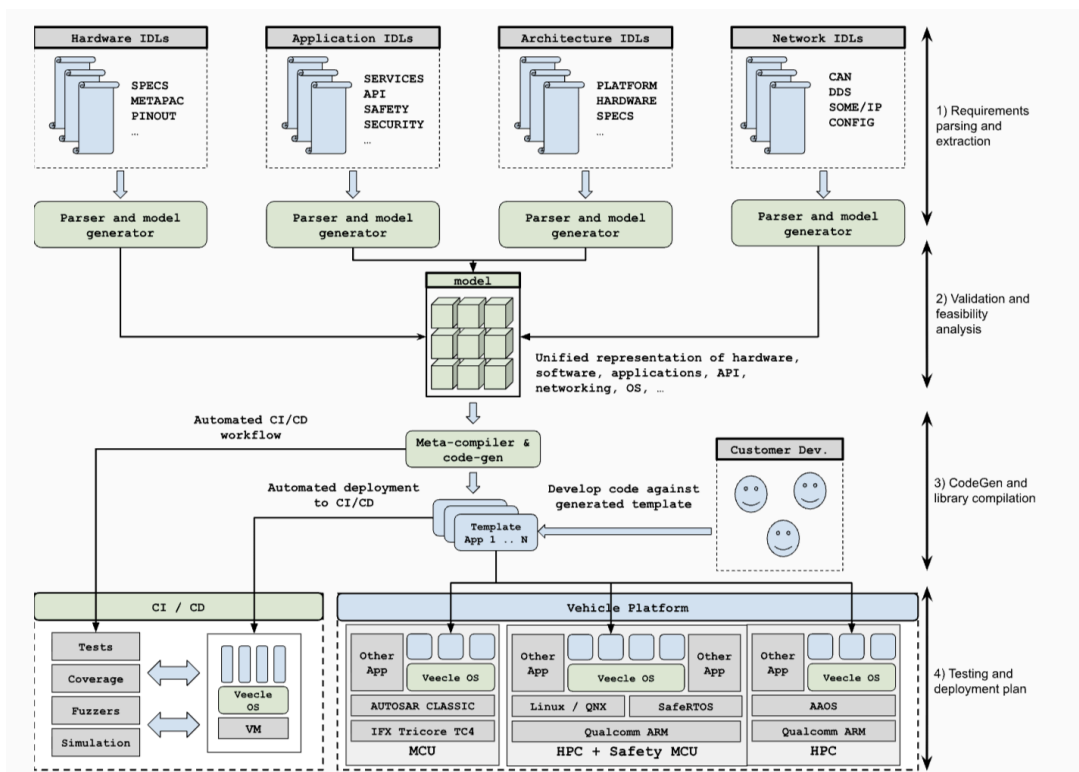
The architecture, hardware, and communication specifications (DBC, DDS IDL, SOME/IP XML, platform specs, safety configuration) are imported into a **unified model**:

- Hardware IDLs → Pinouts, clock domains, peripherals
- Application IDLs → APIs, safety constraints, service boundaries
- Network IDLs → CAN / DDS / SOME/IP configuration
- Architecture → ECU topology and OS capabilities

This unified model (called the **meta-model**) defines:

- Services (bundle of actors) and applications.
- Which data exist and where they originate from, where they need to go.
- Where each Service is allowed to execute (MCU / SoC / HPC / IVI).
- Data dependencies, timing budgets, scheduling requirements.

This model is versioned, validated, and exported as structured data.



3. Meta-Compiler and Code Generation

Our compilation toolchain, **veecle-meta**, uses the model to generate:



- Typed message structures
- Asynchronous handlers and call boundaries
- Deployment configurations (defining which core, process, or task runs each component)
- The correct communication adapters (Store / IPC / Network)

No manual implementation of DDS, SOME/IP, CAN, VHAL, or IPC is required: The Runtime automatically inserts the minimal transport layer needed for each connection.

Most importantly, veecle-meta is plugin-based: it exposes REST-style APIs that third parties can use to interact with both the model and the generated code at any point. This enables seamless integration with external tools and supports straightforward web-based deployments.

4. Deployment Across Targets

The same **WiperLogic App** is deployed into multiple execution domains:

Hardware / OS Target	Role in Application	Veecle Runtime Mode
Safety MCU	Wiper stalks, rain sensor, actuation	no-std async runtime, no heap
HPC Linux Process	Computer vision, inference, logic	Full async runtime with networking
Android Automotive IVI	UI controls, driver preferences	App-layer adapter via VHAL or RPC

The **Orchestrator** ensures communication is transparently managed: No application code changes are required.

5. CI/CD, Simulation, and Verification

Because of the fact that

1. The actors code is target agnostic,
2. The meta-model as full-view of the system, and
3. The meta-model exposes APIs

Testing and analysis become automatic:

Steps	Output
Meta-Model Export → JSON	Full graph of data dependencies & timing constraints.

Static Analysis	Worst-case execution time, dataflow correctness, access ordering.
CI/CD Testing	Compile all actors into x86 mode, isolate each into its own runtime and perform unit tests + fuzzing + virtual simulation.
Hardware Deployment	Attach hardware to CI/CD server and in parallel compile for the target to test hardware features.

Telemetry is automatically collected through structured traces (OpenTelemetry), enabling inspection of dataflow and performance in real time.

Why This Works Across Embedded + HPC

- **One Runtime Model:** async, deterministic, no hidden scheduling behavior.
- **OSAL Layer:** Same API across FreeRTOS, QNX, Linux, bare-metal.
- **Feature-Gated Rust:** No heap in safety domains, full capabilities on HPC.
- **Typed Communication:** Decouples application behavior from protocol or hardware.
- **Orchestrator:** Ensures correct placement, memory visibility, and communication routing.