# Veecle **Development Studio**

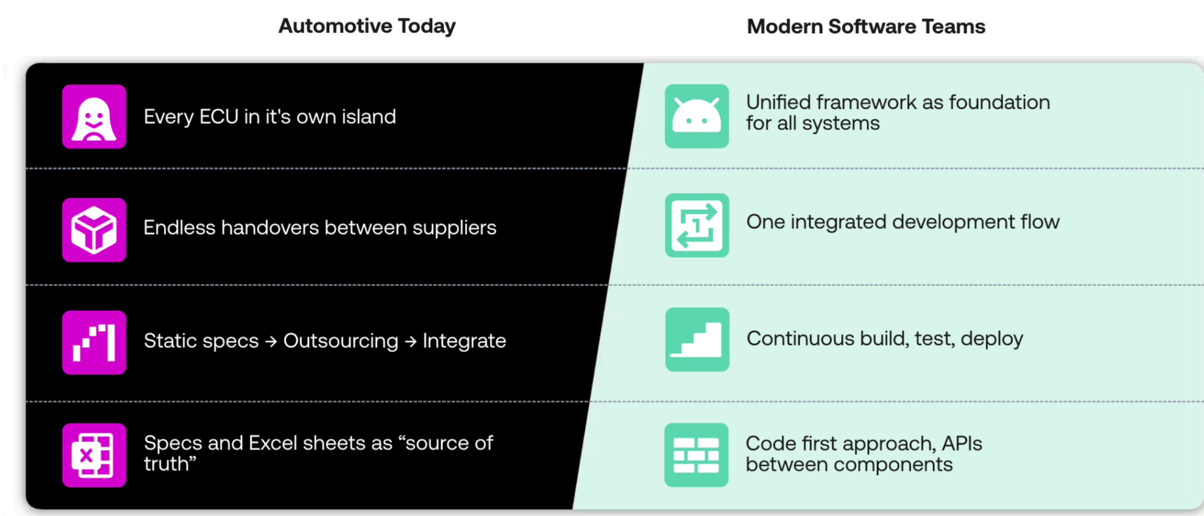
## Technical White Paper v1.0

## Introduction

Over the last decade, consumers have grown accustomed to software that evolves constantly. Smartphones, home devices, and apps improve every few weeks: new features appear overnight, bugs disappear within days, and updates happen almost invisibly in the background. This pace is not accidental—it comes from modern software teams working on a single, unified foundation with continuous testing, automated deployment, and clear APIs between components. These principles allow them to iterate quickly and safely, even on products used by billions of people.

Automotive software, by contrast, grew up under different conditions. Vehicles are long-lived, safety-critical machines with dozens of suppliers, strict validation processes, and hardware dependencies that are unavoidable. The result is a development structure that prioritizes caution and correctness—necessary properties—but in doing so makes software move at the speed of hardware programs rather than the speed of ideas.

The difference is visualized in the diagram below:




The left side reflects a reality familiar across the automotive industry: each ECU has its own toolchain, integration only happens late in the process, and information is scattered across documents, spreadsheets, and supplier interfaces. This structure wasn't a mistake—it was a logical response to decades of increasing complexity, layered supplier ecosystems, and the need for absolute reliability.

The right side shows how consumer software teams operate: one shared framework, one integrated development flow, continuous build-test-deploy, and APIs instead of static documents. These principles dramatically accelerate iteration. Importantly, they are not specific to mobile apps; they are fundamental engineering practices that enable large, complex systems—like cloud platforms, aircraft avionics, or medical devices—to evolve quickly while maintaining high quality.

The automotive industry is now reaching a point where these principles are increasingly relevant. As vehicles become defined by software—personalization, ADAS, connectivity, energy optimization—the ability to iterate rapidly, validate early, and update continuously becomes a competitive necessity. The question is not whether automotive can adopt the speed of consumer software, but how to translate the underlying principles into a domain with far stricter safety, security, and reliability requirements.

Modern tooling, unified frameworks, and code-first APIs make this translation possible. They bring the advantages of fast, integrated software development to automotive in a way that respects functional safety, hardware diversity, and long product lifecycles. With the right foundation, OEMs and suppliers can shorten development cycles, improve quality, and deliver features throughout the vehicle's lifetime—without compromising the rigor that the domain demands.

This white paper is one of two Veecle papers that introduces such a foundation. Veecle OS is the open-source software framework for building modern architectures for critical devices. This paper focuses on Veecle Development Studio, the modern IDE that unifies design, simulation, testing, and deployment of distributed embedded software into a single, integrated workflow. It outlines how software teams can move with the speed of consumer tech while meeting the safety and reliability expectations of the automotive world.

Concretely, it addresses a challenge shared across the industry: developing software centrally while targeting a highly heterogeneous compute platform—from safety microcontrollers to high-performance computers—without duplicating code, rewriting functionality, or introducing integration errors. Veecle Development Studio provides one coherent, model-driven environment where teams can design functionality once, validate it early, and deploy it across all controllers from the same code base, dramatically reducing effort, risk, and time-to-feature.

# Table of Contents



# A Paradigm Shift: Rust and Modern Architectures

To overcome these structural constraints, automotive software engineering must be **iterative**, **collaborative**, and **test-driven**. This requires a fundamental shift in mindset and tooling.

Modern software moves fast because it is built on a clear set of principles:

- code is the source of truth, development is continuous, tooling is unified, and the basics are open rather than reinvented.
- For automotive to benefit from similar speed and predictability, the underlying foundations must reflect these principles as well—adapted for safety-critical systems.

At the core of this shift is a simple idea: fail early, learn fast. Software must be designed, simulated, tested, and validated long before production hardware exists. This requires development environments that are hardware-independent, reproducible, and verifiable by design.

## Rust as the Enabler

The Rust programming language is a key enabler of this transformation. Rust combines the memory safety of high-level languages with the performance and determinism of C—without relying on a garbage collector or runtime. Its compile-time guarantees eliminate entire classes of defects such as null-pointer dereferences, data races, and buffer overflows, which make up the majority of software vulnerabilities and late-stage debugging effort.

Through Rust's strict type system and borrow checker, memory access rules are verified during compilation. Once compiled, these checks disappear, resulting in optimized binaries with zero-cost abstractions: safety without runtime overhead.

In practice, this leads to software that is modular, predictable, and easier to validate—crucial properties for distributed automotive architectures.

## Evolution, Not Replacement

Rewriting entire automotive stacks in Rust would be unrealistic and unnecessary. The practical path is incremental migration: introducing Rust components alongside existing C/C++ modules through the FFI (Foreign Function Interface). This preserves investment in proven legacy code while gradually improving safety, maintainability, and integration quality.

But language choice alone does not unlock the required acceleration. To move at the speed demanded by today's software-defined products, the architecture must reflect a new set of foundations. Veecle's approach is based on four design principles:

1. **Code first.**
   Code becomes the single source of truth. Architecture diagrams, configuration, and documentation are derived from it—not maintained separately. This reduces divergence, errors, and handover friction.
2. **Don't reinvent the wheel.**
   Existing tools, compilers, and workflows remain usable. They integrate as plug-ins rather than being replaced. This reduces disruption and protects OEM investments in current toolchains.
3. **Maximise Developer Experience (DX).**
   Developer experience is not cosmetic—it determines how fast teams can iterate. A unified environment for design, simulation, testing, and deployment reduces context switching and accelerates learning cycles.
4. **Open Source by default.** The foundational infrastructure for modern software architectures is not differentiating. It should be transparent, reusable, and open. Open source provides trust, flexibility, and broad adoption—especially important when code runs across safety controllers and high-performance computers.

## Architecture Must Evolve With the Language

Rust removes many traditional software risks, but rewriting legacy architectures in Rust without changing the underlying model simply preserves old limitations in a new syntax. True progress comes from frameworks that leverage Rust's strengths: reproducible builds, consistent interfaces, deterministic execution, and model-driven validation across heterogeneous hardware.

This combination—a modern language, a unified framework, and principled architectural design—is what finally makes it possible to develop automotive software centrally and deploy it consistently across microcontrollers and high-performance computers from the same code base. It reduces errors, compresses development time, and brings software iteration closer to consumer-grade velocity while respecting the demands of functional safety.

# The Veecle Framework

The described paradigm shift is realized in practice through the **Veecle Framework**, a unified foundation for designing, testing, and deploying vehicle software with the same simplicity as building modern applications.

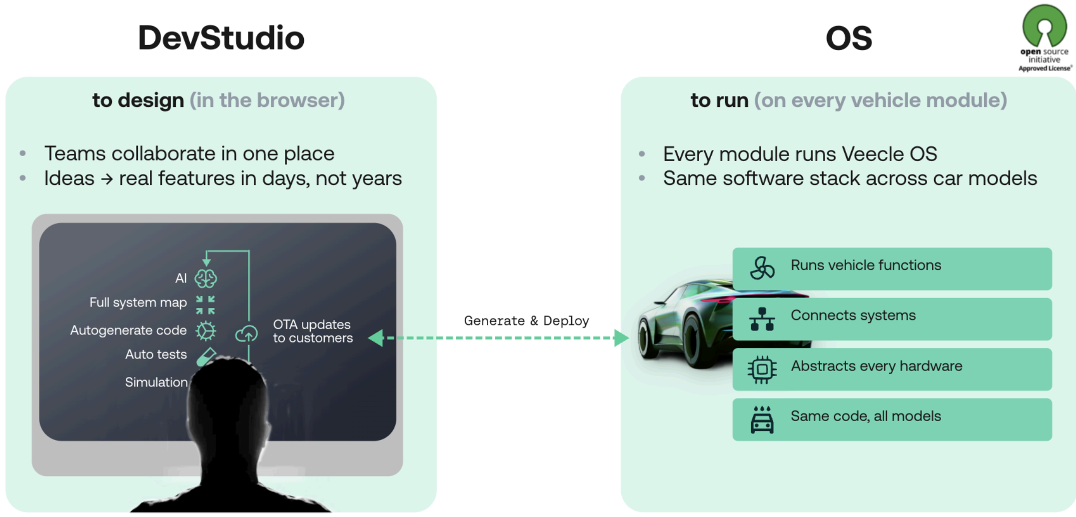
## Architecture Overview

**Veecle Development Studio (Veecle Studio)** and **Veecle Operating System (Veecle OS)** form two complementary components:

- **Veecle Studio** is a centralised server-side SDK that allows engineers to model complete systems, simulate behavior, and execute automated CI/CD pipelines. ECUs, Zone Controllers, and HPCs are represented as part of one integrated system. Developers define *services* that communicate via standardized, typed APIs and distribute them to arbitrary hardware targets. Builds and deployments are fully automated with no local environment required.

- **Veecle OS** is a Rust-based open-source runtime providing a **unified execution environment** across all controller types, from safety-critical microcontrollers running OSEK, FreeRTOS, or PXROS-HR to high-performance compute nodes on Linux or QNX. It abstracts operating systems, middleware, and communication layers, ensuring that the same binary behaves identically across targets.

Together they create a continuous workflow: code and design in the cloud → generate and deploy → run on every controller.



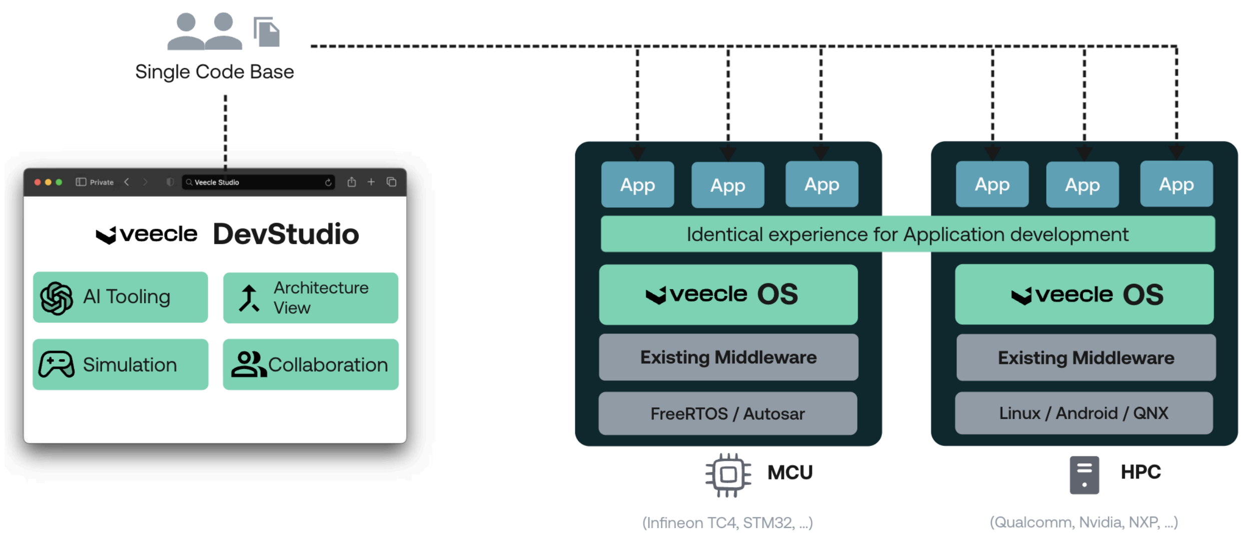

- **MetaModel**: Both components are linked through a formal model that describes the structure and runtime behavior of the system, including communication channels, hardware mappings, configuration, constraints, and safety parameters. We call this the "MetaModel", and it acts as the single source of truth for the entire development, build, and deployment pipeline.

- **CLI Toolchain**: The command-line utility ***nmm*** transforms the MetaModel configuration (TOML) into consistent build processes, target binaries, test artifacts, and deployment scripts. It orchestrates Rust's cargo, handles dependency resolution, and ensures reproducibility across multiple operating systems and processor architectures. Veecle OS can thus be used entirely standalone, ideal for series production or developers preferring CLI workflows.

- **Toolchains and Compilers**: Veecle supports standard LLVM-based toolchains by default but allows easy substitution with **safety-certified LLVM compilers** that support both Rust and C/C++. This provides compatibility with ISO 26262 up to ASIL D, allowing OEMs to merge existing C/C++ modules with new, memory-safe Rust components in a unified build environment without changing the process or build system.

The result is a consistent, hardware-independent development process where the same tools, languages, and methods can be used from early simulation to production deployment.

## Generic Workflow




The Veecle workflow follows a clear and repeatable sequence applicable to both AUTOSAR-based and lightweight FreeRTOS setups:

1. **Modeling** – Define system architecture and service interactions in the Studio using visual editors or code.

2. **Simulation and Mocking** – Replace unavailable drivers or ECUs with mock services; validate interactions early.

3. **Continuous Integration** – Automatically build, test, and deploy through CI/CD pipelines; validate each iteration.

4. **Hardware Deployment** – Flash identical binaries to real targets (e.g., Infineon AURIX, STM, NXP) without code changes.

5. **Safety Migration** – Replace the runtime (FreeRTOS → SafeRTOS or PXROS-HR or AUTOSAR or QNX) without modifying application logic.

6. **Runtime Observation** – Annotate code with **#[veecle::instrument]** for real-time tracing of tasks, latencies, and dataflows inside the Studio.

7. **Scalability** – Use the same architecture across MCU, ZC, and HPC layers; Veecle OS automatically selects the most efficient data path (zero-copy, IPC, or serialized transport).

This unified process combines modern software agility with the determinism and traceability required in safety-critical systems.


# Example: AUTOSAR Integration (Built on the Generic Workflow)

Veecle OS can be integrated into **AUTOSAR Classic** as a standard software component (SWC) or separate task, executing new Rust-based logic while maintaining compatibility with existing C modules and toolchains.

- **Purpose**: Enable faster development, memory-safe logic, and modular code without invalidating the existing AUTOSAR certification base.

- **Operation**: Veecle OS runs within its own tasks, interfacing with other components via the RTE. The established AUTOSAR infrastructure—MCAL layers, drivers, and safety artifacts—remains untouched.

- **Isolation (Method 1)**: Rust services can run in separate processes or address spaces. C modules remain *Safety Elements out of Context* (SEooC), preserving *freedom from interference* and all existing safety certificates. Communication occurs via RTE, shared memory, or message passing, so unsafe C code cannot compromise Rust's safety guarantees.

- **FFI Integration (Method 2)**: Where strict isolation is impractical, Veecle supports Foreign Function Interface (FFI) bindings. Rust code can directly call existing C libraries, similar to header inclusion and linker binding. However, this should be reserved for controlled cases since C-side memory errors (e.g., use-after-free, out-of-bounds) bypass Rust's compile-time protection.

**Result:** a hybrid environment combining the reliability of C/C++ with the safety and modularity of Rust. The same LLVM-based toolchain (open or certified) builds both, ensuring consistency across environments.

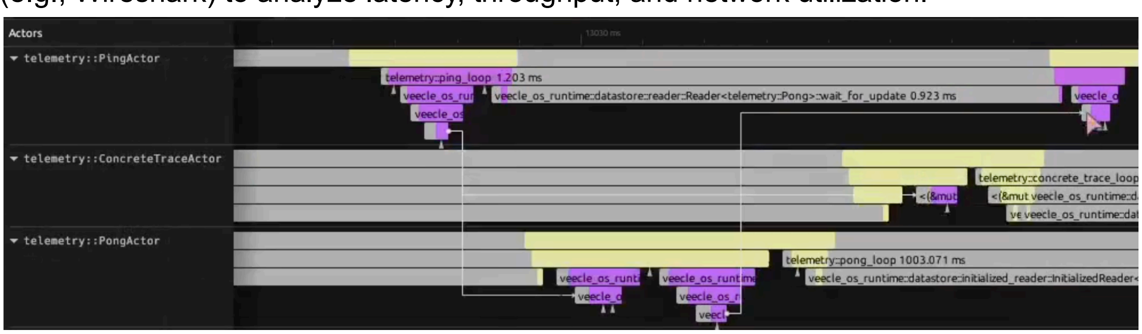
## Example: Lightweight FreeRTOS Setup (AUTOSAR-Free, Code-Equivalent)

For non-AUTOSAR environments or early development stages, Veecle OS runs as a task under **FreeRTOS** or **Embassy**. This configuration is the default in Veecle's Starter Kit and supports both rapid prototyping and QM-grade production.

- **Application Compatibility**: The same application logic and service graph used in AUTOSAR can run unmodified under FreeRTOS. Later migration to SafeRTOS, PXROS-HR, or QNX requires replacing only the runtime, not the business logic.

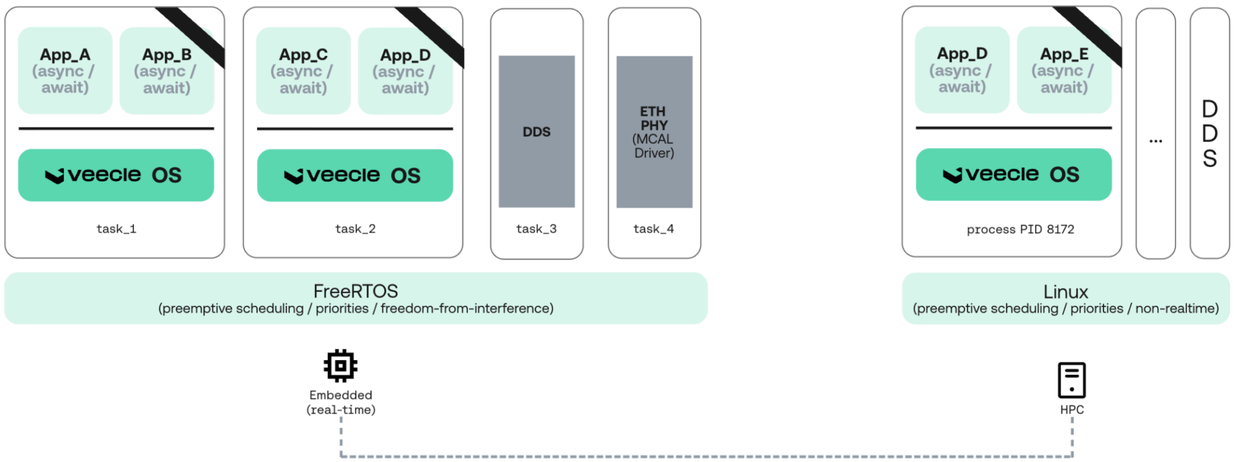
- **Service Orchestration**: Veecle OS manages multiple independent microservices with high cohesion and minimal coupling. Communication between services uses thread-safe FreeRTOS queues or standardized network protocols such as DDS, SOME/IP, or Zenoh.

- **Observability**: Dataflow and timing behavior can be inspected live using standard tools (e.g., Wireshark) to analyze latency, throughput, and network utilization.




- **Hardware Independence**: Applications can be validated virtually long before target ECUs exist. Mock services emulate hardware interfaces, drastically reducing testing time and integration risk.

This setup allows developers to build and validate full vehicle architectures early and scale them seamlessly to production environments—without changing a single line of application code.




# Resulting Impact

Modern software frameworks like Veecle not only transform *how* vehicle software is built but also *who* controls it and *how* it evolves over time.

Instead of closed ecosystems, vendor lock-in, and opaque toolchains, Veecle provides an **open, transparent foundation** on which manufacturers can design and extend their own architectures.

## Open Source by Design

The open-source nature of **Veecle OS** is not a marketing claim but a deliberate architectural decision.

The runtime is published under the **Apache 2.0 license**, approved by the Open Source Initiative (OSI) and explicitly permitting commercial use.

This ensures long-term availability independent of corporate ownership, product cycles, or license models. OEMs and Tier-1s can inspect, modify, or extend the source code themselves.

Unlike closed systems that hide vulnerabilities behind NDAs and vendor agreements, Veecle's transparency allows early detection and correction of security issues by a growing developer community.

This openness also aligns naturally with upcoming regulatory frameworks such as the **EU Cyber Resilience Act (CRA)**, which demands verifiable software supply chains and proactive vulnerability management.

By using open, reviewable components (including compilers based on LLVM technology), Veecle helps companies meet compliance requirements without additional overhead.

This model builds trust: every organization can see the exact foundation their safety-critical systems depend on, validate it independently, and remain confident that the codebase will still compile and evolve a decade from now.

## Architecture Ownership and Extensibility

**Veecle OS** is not a fixed product but a **construction kit** for defining and maintaining one's own software architecture.

System architects start by designing **high-level APIs** for key vehicle domains. For example:

None

```
headlights.dipped_beam()
headlights.high_beam()
headlights.indicator()
```

These abstract APIs serve as the shared language between teams.

Application developers work exclusively against these interfaces, without direct exposure to CAN messages, I/O drivers, or RTOS task scheduling.

Beneath the API layer, hardware-specific components can be optimized, replaced, or ported to new processors or OSes without changing a single line of application logic.

This separation of **application**, **architecture**, and **infrastructure** makes development highly parallelizable and verifiable.

Multiple teams can develop and test independently while maintaining consistent integration through stable, versioned APIs.

Integration shifts from a late-stage milestone to a **continuous process**, significantly reducing late rework and enabling incremental delivery.

## A New Role for Suppliers

This decoupling also redefines the position of suppliers.

Instead of shipping entire ECUs or proprietary binaries, suppliers can provide **independent, digitally signed software modules** that integrate into Veecle OS as *Software-as-a-Product (SaaP)*.

OEMs remain in control of configuration, updates, and diagnostics, while suppliers retain ownership of their intellectual property through **signed and encrypted binaries**.

Only the OEM's CI/CD pipeline can integrate these modules, guaranteeing authenticity and integrity.

Because Veecle abstracts hardware, middleware, and OS dependencies, the same signed component can run safely across multiple platforms without code changes.

This creates an **open ecosystem** that encourages collaboration and reuse instead of duplication.

Knowledge becomes shareable, integration becomes simpler, and innovation accelerates across organizational boundaries.

## Conclusion — The Auto Industry's Opportunity Through Modern Software Foundations

The transition toward software-defined vehicles is not a temporary trend, it is a structural transformation.

The automotive industry faces the same turning point that the telecom sector did two decades ago: **only those who master software will remain independent**.

True sovereignty will not come from proprietary middleware but from **clean, auditable, and open software foundations** on which manufacturers can build their own architectures. Openness, safety, and speed are not trade-offs, they reinforce each other.

Veecle exemplifies this principle. Built entirely in Rust, guided by a formal MetaModel, and maintaining a clear separation between architecture, application, and hardware, Veecle combines **safety, agility, and transparency** in a single framework.

Developers create modular functional blocks that can be orchestrated across all ECUs, Zone Controllers, and HPCs, supported by modern LLVM-based toolchains that unify Rust and C/C++ for mixed-language systems.

The same design philosophy applies far beyond automotive: agriculture, robotics, industrial automation, and defense all share similar demands for safety-critical, updateable software. The long-term vision is a **shared automotive software foundation** for all intelligent machines, transparent, certifiable, and free from vendor dependency. A foundation that accelerates innovation, fulfills regulatory expectations, and secures technological sovereignty.

A foundation for an industry that **builds** again rather than merely **assembles**.