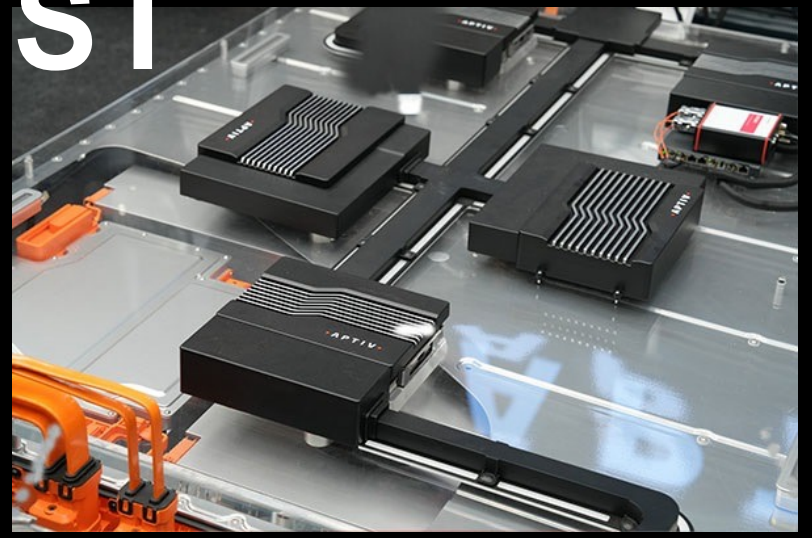
April 26, 2026

Christof Petig
Staff Embedded SW developer

INSERTING WEBASSEMBLY BETWEEN C++ AND RUST FOR FUN AND PROFIT



APTIV

TL;DR

This is an ideal solution for

- Combining many languages
- Combining binaries
  - Stable ABI bridges incompatibilities
  - wasm Sandboxing for untrusted binaries or FuSa

But it gets difficult for

- Threads, zero copy, generics, object inheritance
- Passing pointers & callbacks
- Bidirectional dependencies

About me

Born 1970, Diploma in microelectronics,
wife (SMB consultant),
two kids (chemical scientist, psychotherapist)

Prog. Languages: BASIC, Assembler, C, C++, Rust (…)

Textile industry: ERP & Co

Automotive: ADAS, AI acceleration, Rust, WebAssembly

Organizations: SAE, AUTOSAR, ByteCodeAlliance

Hobbies: Acting, Cycling

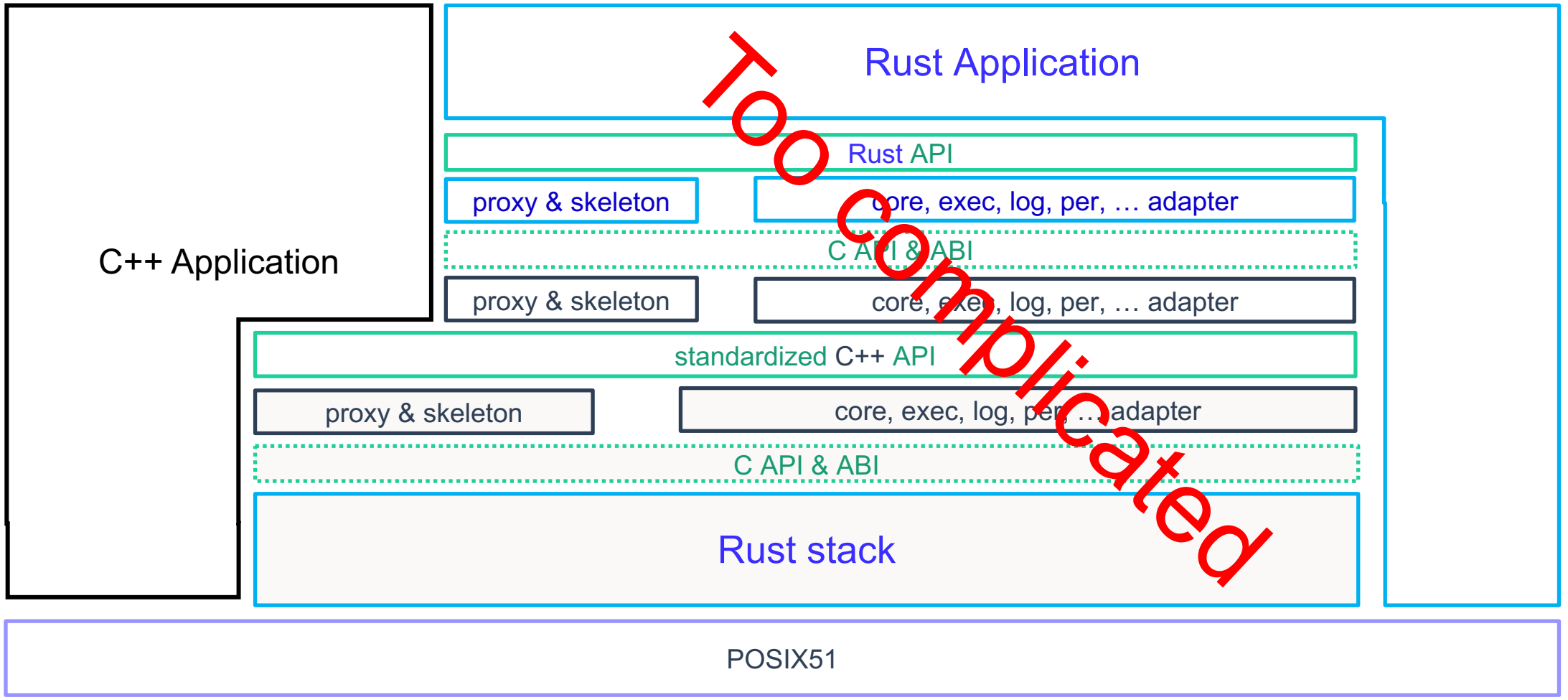
Structure

1. Motivation and background
2. WebAssembly + Component model intro
3. Symmetric ABI
4. Technologies to expect

The bigger picture

AUTOSAR Adaptive Platform Stack

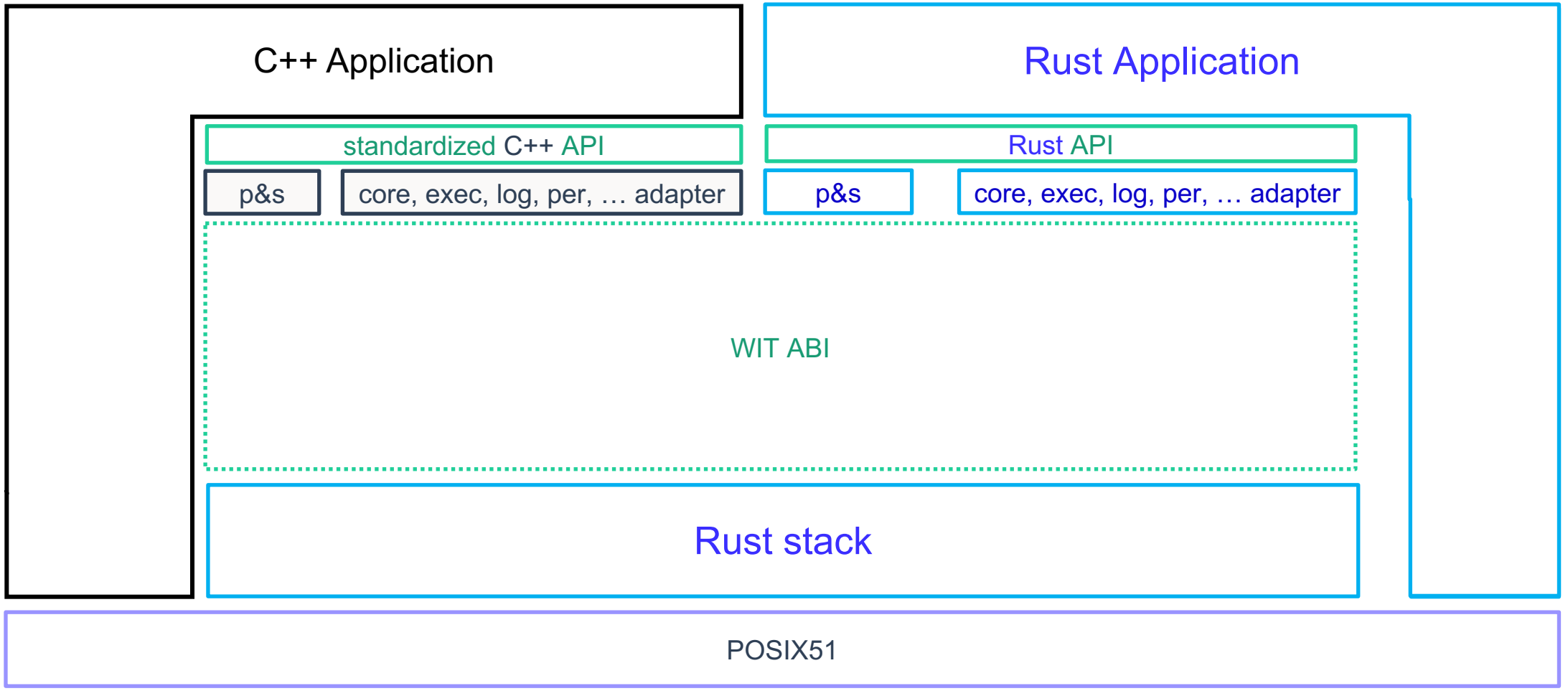
Adding Rust and language adapters via C



APTIV

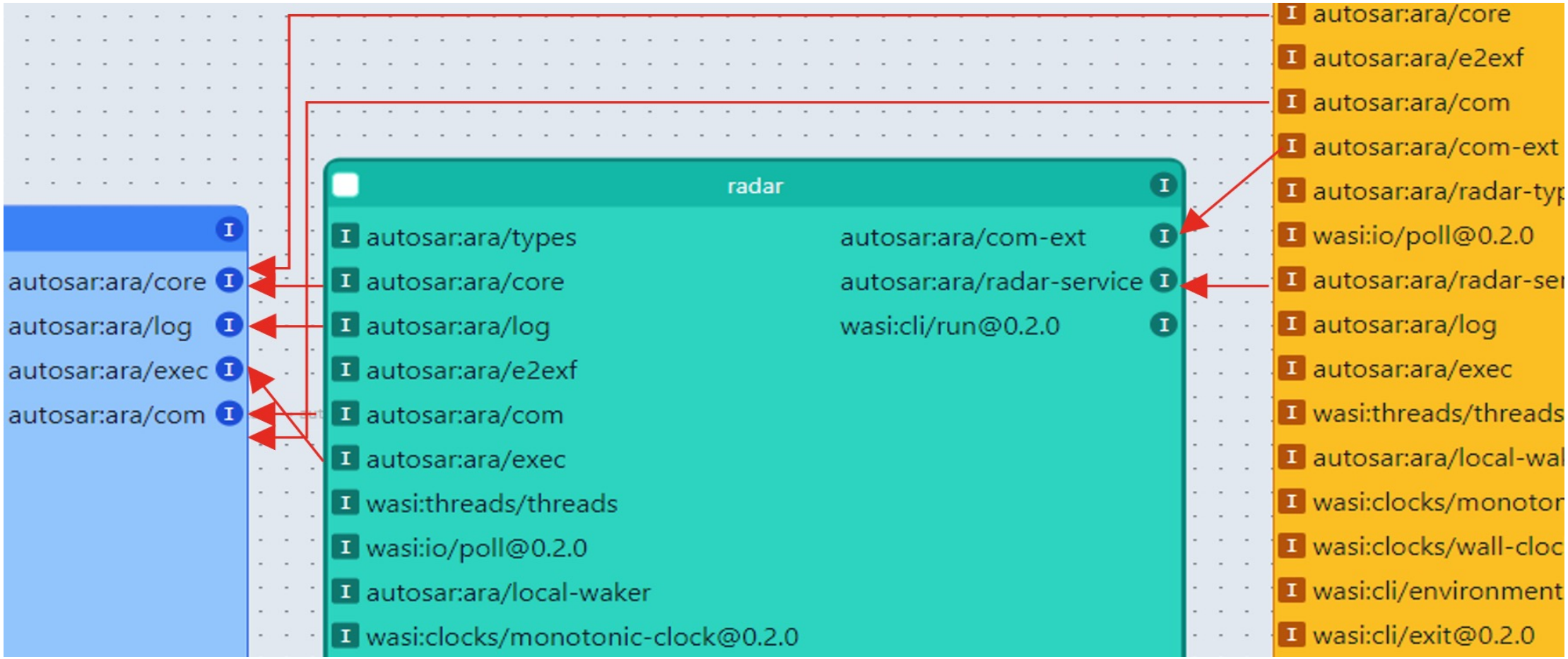
AUTOSAR Adaptive Platform Stack

WIT without WASM



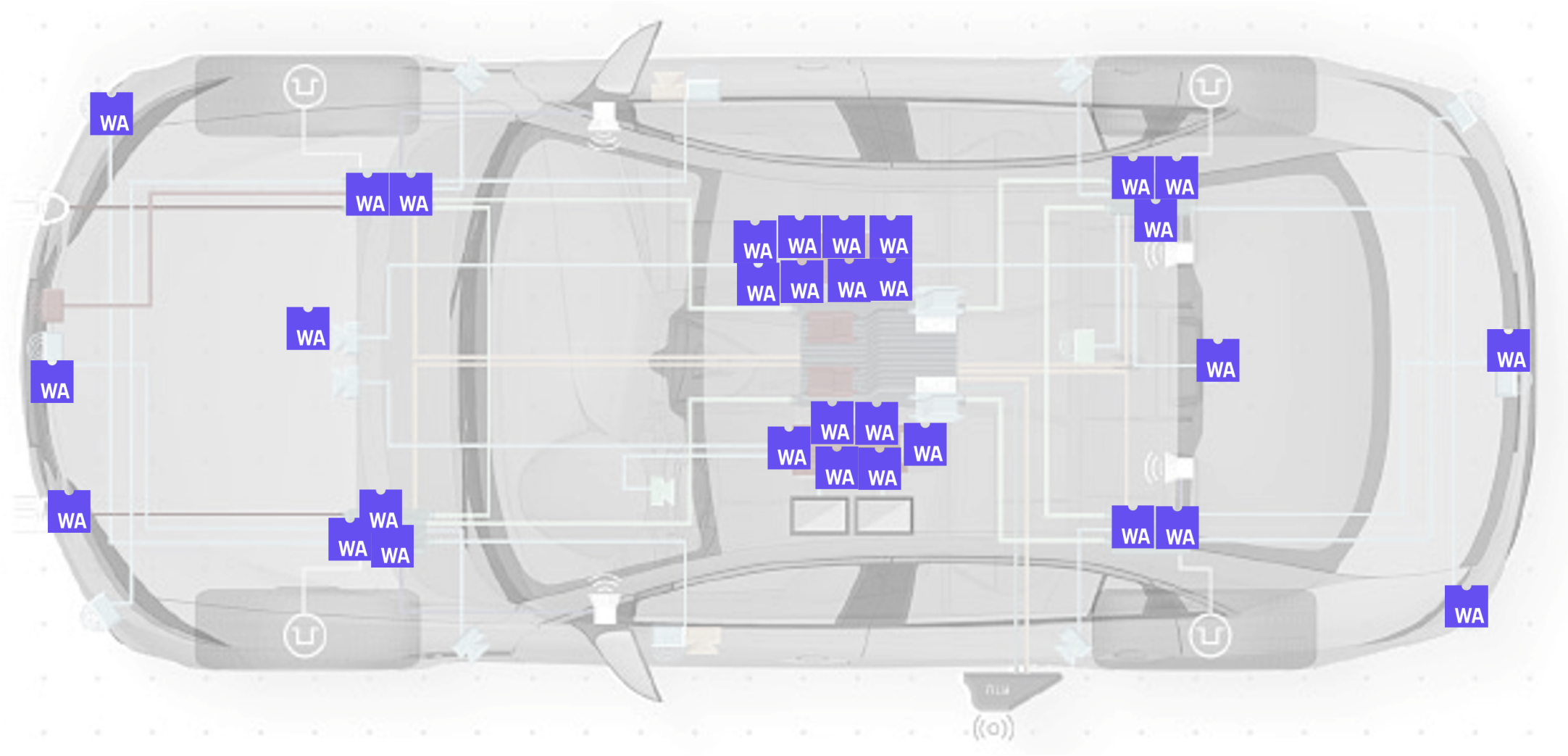
Zooming in

AUTOSAR adaptive (ara) as well as Operating system (wasi) APIs, versioned



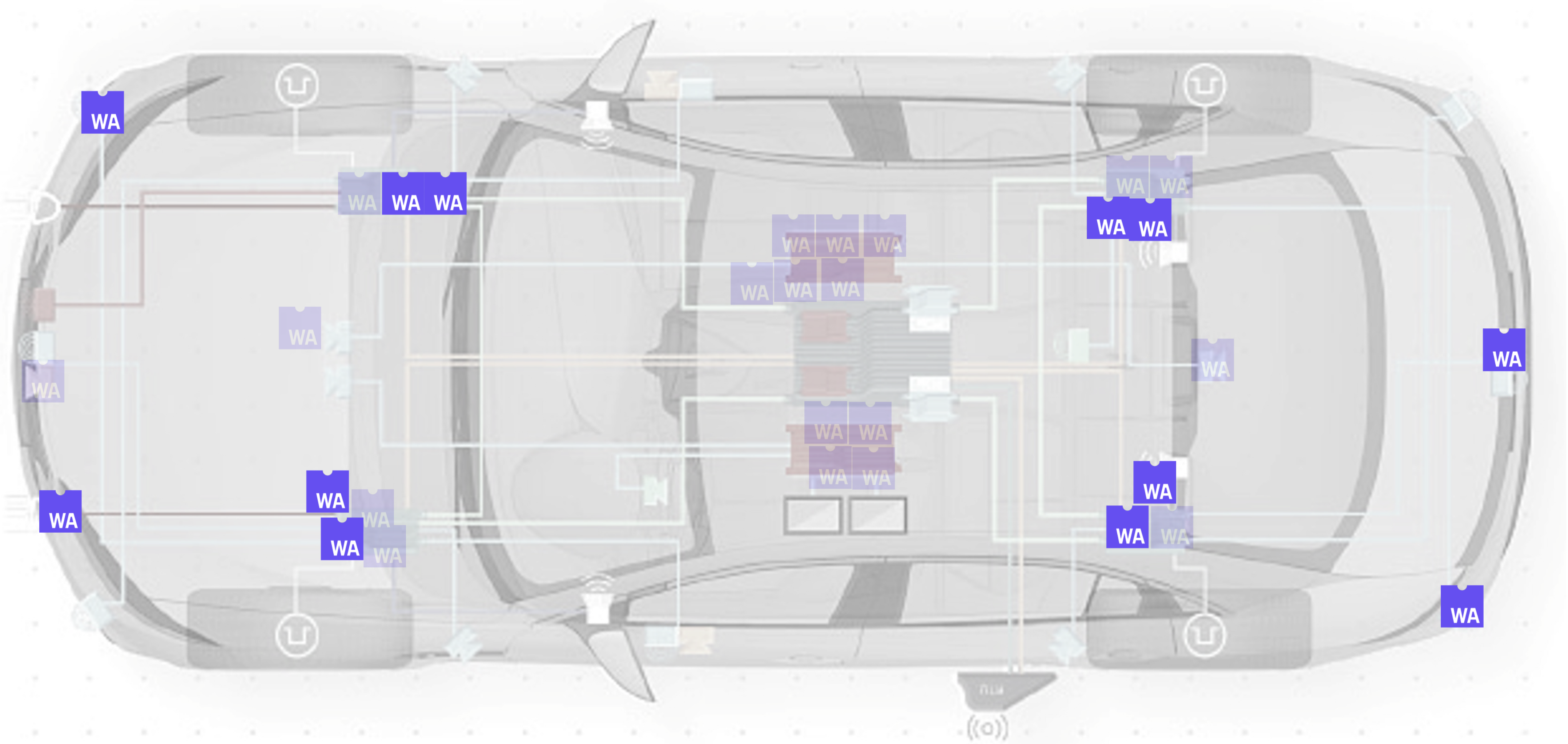
APTIV

Wasm components fit anywhere



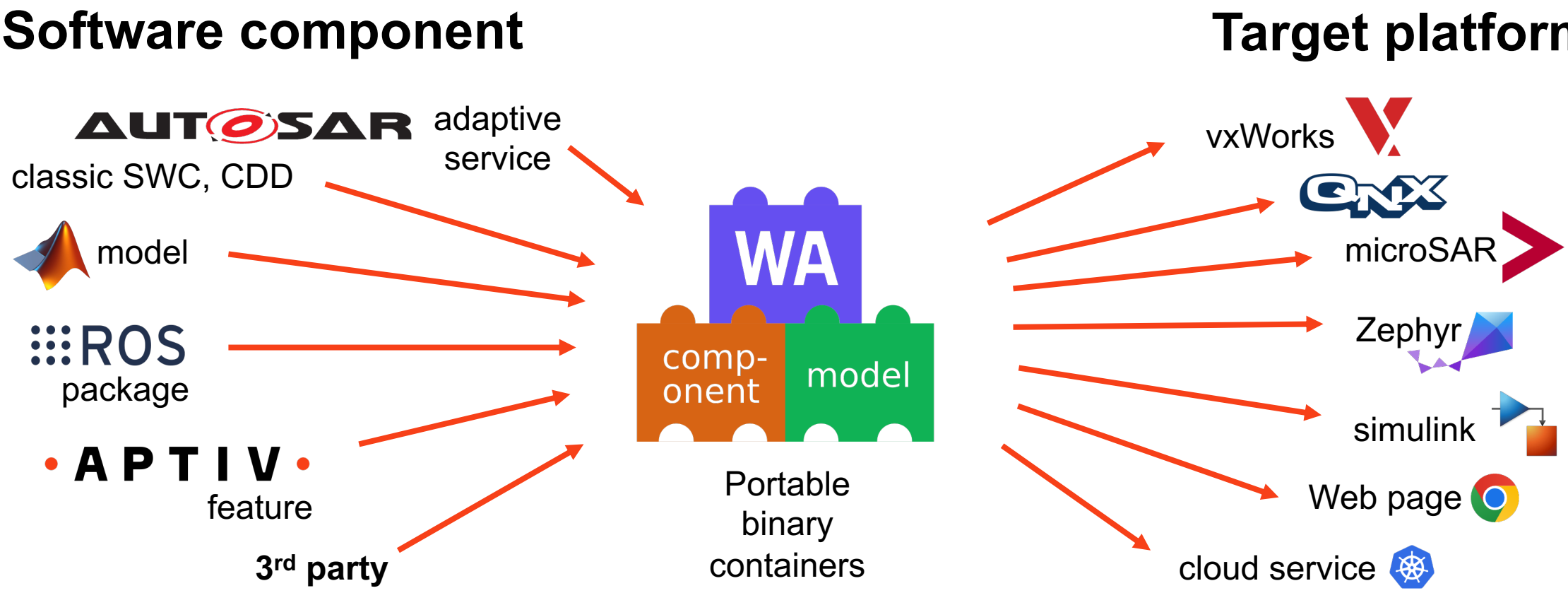
APTIV

You can move them on powerdown



APTIV

Unify input environment, deploy with flexibility



APTIV

Component model deep dive

WebAssembly

W3C standard

compile C programs for the browser

(multi-threading was a weak point from the beginning)

A well-defined virtual CPU target with dedicated linear memory which can be translated to real CPUs in a single pass

Emscripten started compiling to asm.js, then wasm

threads are emulated with web workers and java script glue

Rust uses wasm-bindgen for this target

Downside: wasm for the browser not in the server

Component model

A different ecosystem (not emscripten nor wasm-bindgen!)

The WebAssembly standard behind WASI (System Interface) 0.2+

Defines representation of higher-level data types (canonical ABI)

- how to copy data in and out of a sandbox
- shared nothing

A new binary format

- multiple modules (binaries) with “signal” connections
  - and external interface definition
- conceptually like a docker container

WebAssembly details

History of the CM

2015 WebAssembly: Run C in the browser more efficiently than asm.js

2019 WASI 0.1: Run wasm on a server

2025 WASI 0.2: More modular, less POSIXy, “how to pass a string?”

2026 WASI 0.3: Async

Features of the CM

Sandboxing, language and OS agnostic interfaces

Formerly called **Interface Types**

Webassembly Interface Types (WIT)

WASI 0.1 started with C headers, then used witx (S expression)

WASI 0.2 standardized WIT

Data types:

u8, s8, … u64, s64, tuple, variant (option, result), string, list, record,

resource (remote object)

Functions, interfaces, worlds

WASI 0.3 adds async functions (implicit future), stream, future (object)

+ cooperative threading (waitable sets)

WASI 0.2.11 0.3.0-rc-2026-03-15

Subsystems

- **random**
- **clocks**
- current, timeout
- **cli**
- environment, main, exit, stdio, terminal
- **http**
- service, client, request
- **filesystem**
- **sockets**

Canonical ABI

- Arguments on stack vs in memory threshold
- Mechanism to allocate and free from linear memory
- 0.3: Conventions for async waiting & callback

CM call in detail (string arg, string result)

Host calls cabi_realloc to allocate in linear memory

Host passes address and length to guest function

Guest puts string somewhere (malloc?)

Stores address and length in static memory

Returns address

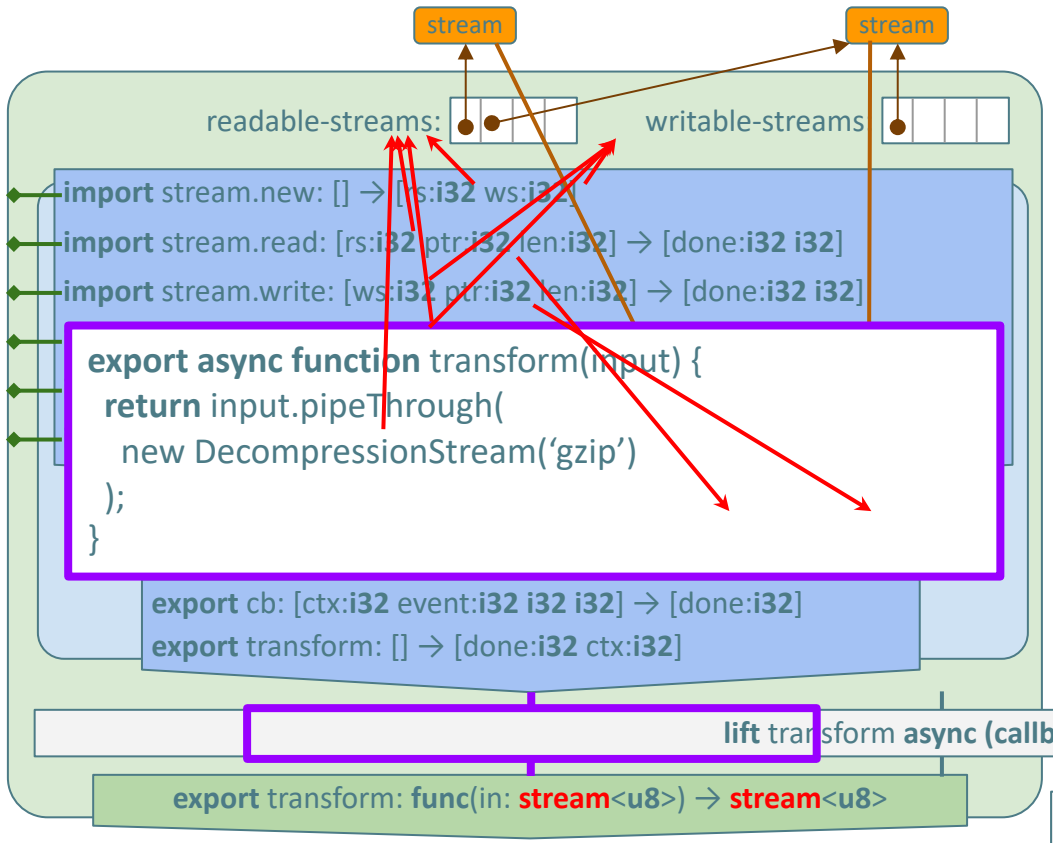
Host processes string and calls cabi_post_F

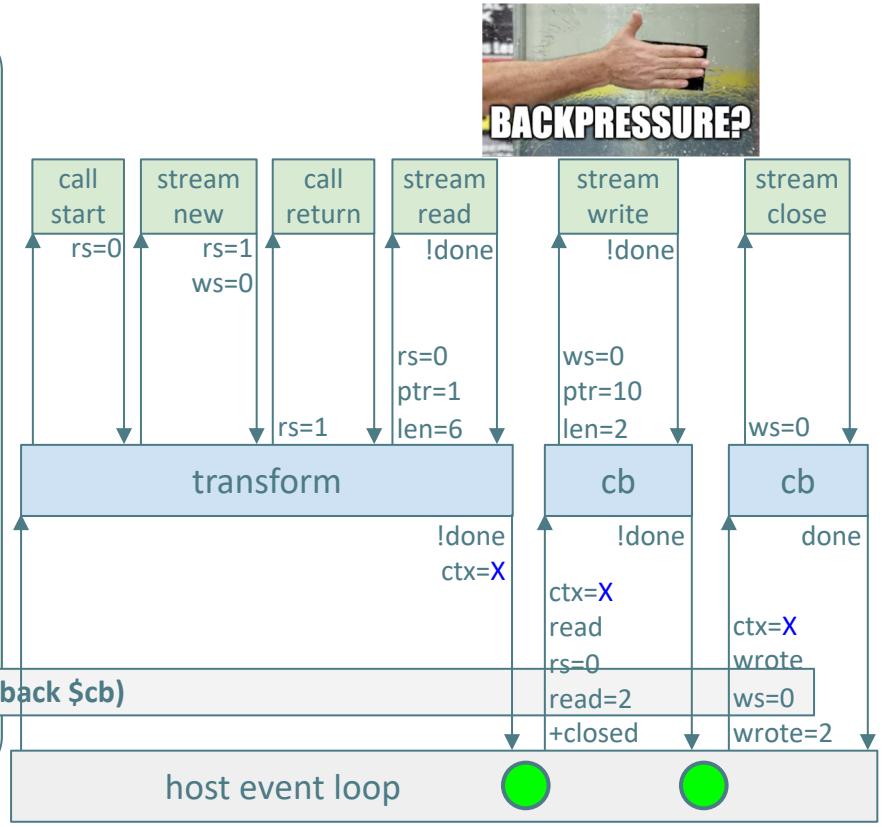
Stream in detail (by Luke Wagner)

completion based (like io_uring)

designed for language integration

Sample of streaming





APTIV

Symmetric ABI

- Inspired by guest to host calls
- Callee exports the interface expected by the caller, no host intermediate
- Argument Strings are borrowed, results are moved
- More complex resource and stream changes

Avoiding allocations

Different approaches:

- **Caller provided buffers**
- **lazy lowering**
- **shared memory (publisher subscriber)**

WebAssembly in automotive

- **AUTOSAR adaptive uses WIT/symmetric**
- **The dream of portable binaries (OS + arch agnostic)**
- **VSS/uProtocol WASI equivalent needed**
  see e.g. vspec2wit (ETAS, COVESA 2023)

Outlook

Cooperative threads

Zero copy, zero allocation, publisher subscriber

Shared everything threads

Personal Todo

- Merge C symmetric async upstream
- Document symmetric mode in upstream
- Standardize shared memory
- Undust Rust C++ AUTOSAR adaptive example

Resources

https://bytecodealliance.org/

https://github.com/cpetig/wit-bindgen symmetric fork

https://wasmcon24.sched.com/event/1qvIG/component-model-in-software-defined-vehicles-christof-petig-aptiv symmetric presentation (video on YouTube)

Thank you.

Additional resources

Functional safety and AoT compilation

Multi-language MC/DC:
https://github.com/pulseengine/witness
Cortex-M AoT:
https://github.com/pulseengine/synth

ByteCode Alliance

Non-profit organization for the development and promotion of the component model

Hosts the source code, web sites, etc.

Funds strategic development and legal counsel