Towards a Pattern Language for Mixed Microprocessor and Microcontroller (Hybrid) Edge Development

Hugo Sereno Ferreira
hugo.ferreira@arm.com
Arm
Cambridge, UK

Basma El Gaabouri
basma.elgaabouri@arm.com
Arm
Cambridge, UK

Chris Adeniyi-Jones
chris.adeniyi-jones@arm.com
Arm
Cambridge, UK

Eric Van Hensbergen
eric.vanhensbergen@arm.com
Arm
Austin, Texas, USA

ABSTRACT

Nowadays, many System on a Chip (SoC) designs, particularly those targeting Edge computing, incorporate multiple microprocessors, microcontrollers, and specialized accelerators for highly efficient, low power functions (Hybrid Systems). Developing and deploying for these systems offer significant versatility but also present unique challenges, such as resource allocation, differences in development tooling, portability, programmability and discoverability. In this paper we landscape the concerns and forces that shape the design decisions for hybrid Edge systems, by proposing eighteen (18) patterns arranged in a pattern language. These patterns, derived from literature review and empirical experience emphasise the increasing relevance of cloud-native practices. We also identify some open challenges that represent key areas for future research, including further concerns on portability and development of advanced tools for managing the software lifecycle at the Edge.

CCS CONCEPTS

• **Computer systems organization** → **Architectures**; **Embedded systems**; **Real-time operating systems**; • **Software and its engineering** → **Design patterns**; **Operating systems**.

KEY WORDS AND PHRASES

Computer Architecture, Edge computing, Internet-of-Things, Distributed Systems, Patterns

ACM Reference Format:
Hugo Sereno Ferreira, Basma El Gaabouri, Chris Adeniyi-Jones, and Eric Van Hensbergen. 2024. Towards a Pattern Language for Mixed Microprocessor and Microcontroller (Hybrid) Edge Development. In *Proceedings of 31st Conference on Pattern Languages of Programs, People, and Practices (PLoP 2024).* ACM, New York, NY, USA, 12 pages.

Proceedings of 31st Conference on Pattern Languages of Programs, People, and Practices (PLoP 2024), 2024.
PLoP 2024, October 13–16, 2024, Skamania Lodge, Columbia River Gorge, Washington, USA.

Hillside ISBN 978-1-941652-20-6

1 INTRODUCTION

The definition of Edge Computing is constantly evolving due to ongoing technological advancements, shifting industry requirements, and changes in the broader landscape of computing and data management. For instance, the rise of Internet of Things (IoT) devices has significantly pushed the development of edge computing, as these devices generate massive amounts of data that need to be processed close to their source to reduce latency and bandwidth usage [10]. Several reports indicate that edge computing will become a critical factor in the deployment of next-generation applications, including autonomous vehicles, smart cities, and industrial automation. Although the concept of edge computing has been fluid and subject to debate since the early days of its adoption [12], the fundamental principles and goals of Edge Computing have remained relatively stable, focusing on processing data closer to its source to improve speed, efficiency, and reliability.

Many System on Chip (SoC) designs now incorporate both Cortex-A and M processors[1], along with specialized accelerators for functions such as machine learning. This integration within a single SoC provides high processing power and diverse functionality, making these chips particularly suitable for a variety of applications. These include industrial automation, automotive systems, energy infrastructure, smart home technologies, and entertainment devices. Each of these areas benefits from the advanced processing capabilities and specialized functions of these SoCs, highlighting their versatility and value in modern technological devices and systems.

In many practical applications, Cortex-A cores run a feature-rich operating system (often Linux) to provide a user-friendly interface with display and interactive components for an intuitive user experience. Meanwhile, the M cores operate a real-time operating system, such as FreeRTOS [3] or Zephyr [7], to ensure *timely* processing and minimal latency for *critical tasks* requiring real-time performance. Accelerators designed to enhance computational tasks are typically[2] coordinated by applications running on the Cortex-A cores. These applications manage the accelerators' operations, enabling them to function more effectively within the system.

[1] As this research was conducted at Arm, we make free usage of the typical classification for our products, including exhaustive usage of Cortex-A for referring to application-class CPUs, and Cortex-M for microcontrollers. Nonetheless, we believe the reader would be able to easily translate these references to their particular use case.
[2] Though increasingly non-exclusively.

Some hybrid SoC's incorporate multiple Cortex-M cores and assign/distribute different responsibilities to enhance the system's overall functionality. Certain M cores may be dedicated to specific tasks, such as managing tome-sensitive interactions with sensors and actuators. Other cores might handle software-controlled wireless communications, providing efficient and flexible networking capabilities. Additionally, some may be used for "general-purpose" computing, managing a variety of tasks and operations to support the system's applications and processes. This combination of specialized and general-purpose cores allows hybrid SoCs to operate more efficiently than a single general purpose Cortex-A.

There are numerous motivational use cases for these hybrid systems. One notable example involves balancing energy consumption and performance by selectively activating parts of the system based on application needs. For applications that do not require constant functionality, most of the system can enter a sleep mode during idle periods, with only the M cores and necessary sensors or inputs remaining active. In this scenario, the M cores can wake the rest of the system when needed, such as in response to a specific stimulus like the detection of a *"wake word."* This approach optimizes energy efficiency while maintaining performance when required.

As a result, software developers for Edge Computing now face a significant increase in complexity, primarily due to the expanding diversity of hardware configurations. This is on top of previous challenges developers working on such system had to traditionally take into account, namely the variety and fragmentation of devices with different operating systems, communication protocols, and processing capabilities. Multiple case studies by Amazon Web Services (AWS) illustrates these challenges, showing how developers need to integrate a variety of technology within cohesive ecosystems [1, 6, 8]. As the variety of hardware configurations continues to grow, developers must adapt to a wide range of systems and configurations, such as integrating accelerators for AI processing, ARM processors for energy-efficient tasks, and FPGAs for custom and highly specialised workloads. This presents a complex challenge that demands innovative approaches and solutions to effectively manage and optimize Edge computing environments.

Significant differences exist between the portable, frictionless container-based development model and traditional embedded systems development flows. Container-based development is designed for ease of transfer and implementation, reducing friction in the development process. In contrast, traditional embedded systems are highly optimised for specific SoC configurations and boards, tailored to deliver maximum efficiency and performance for particular hardware setups. Navigating these differences requires a deep understanding of both models and the ability to balance their benefits within a given project.

1.1 Target Audience

The target audience of this paper includes system developers focused on hybrid edge systems who wish to adopt a modern, cloud-native approach to managing the software lifecycle. Application developers will also find value in the described patterns, as they can aid in designing applications that effectively leverage underlying system capabilities and reduce friction in adopting these approaches. Additionally, researchers in this field may be interested in the gaps and challenges identified towards the end of this article, providing insights into potential areas for further investigation and development.

1.2 Paper Structure

The remainder of this paper is structured as follows. Section 2, *"Background,"* identifies the common characteristics of hybrid edge systems and delve into the challenges that typically arise in these scenarios. Section 3, *"A Pattern Language,"* presents eighteen (18) patterns we believe form an initial foundation for a pattern language targeting hybrid edge systems. We conclude our paper in Section 4, *"Conclusions,"* by summarising the key insights and contributions.

2 BACKGROUND

In this section, we provide a general overview of hybrid edge systems, highlighting their common characteristics and the unique challenges they present. Hybrid edge systems integrate various processing cores and specialized components, enabling high performance and versatility. In doing so, these systems face significant challenges derived from their resource constraints, real-time performance requirements, and the complexity of managing diverse hardware components, and highly fragmented software stacks. We will also discuss the typical Cloud-native philosophies for managing deployments and explore how these approaches can be adapted and translated to the Edge.

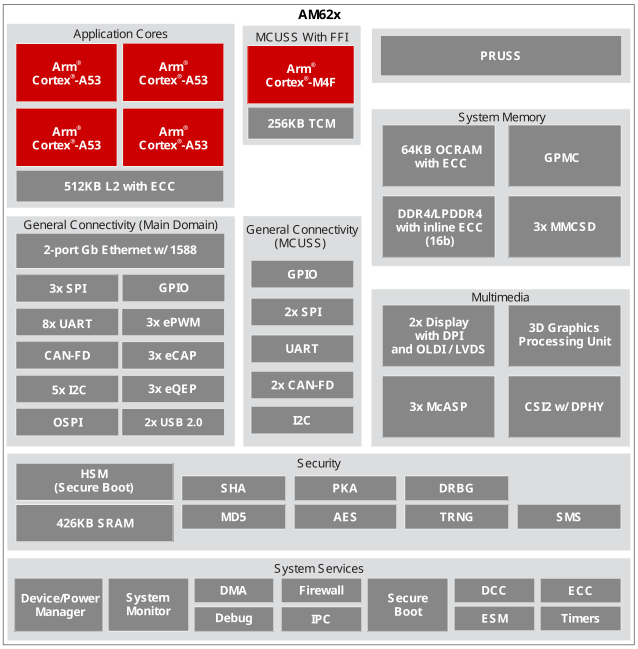

Figure 1: Block diagram of the Texas Instruments AM625 used on the BeaglePlay board, which include a quad-core Cortex-A53 processor and a Cortex-M4F on the same SoC.

An example of a board that we would classify as comprising an hybrid system is the BeaglePlay board, that contains two SoCs, *viz.* (1) a Texas Instruments AM625 with a quad-core Cortex-A53

application processor and a Cortex-M4F, for which `remoteproc` (*cf.* Section 3.7) drivers exist and it is the standard way of programming the latter (*cf.* Figure 1), and (2) a Texas Instruments CC1352 with a user programmable Cortex-M4F, and a RF dedicated Cortex-M0 (*cf.* Figure 2). Programming this particular CC1352 usually involves a less standard approach compared to `remoteproc`, requiring custom software that sends the payload via a serial interface.

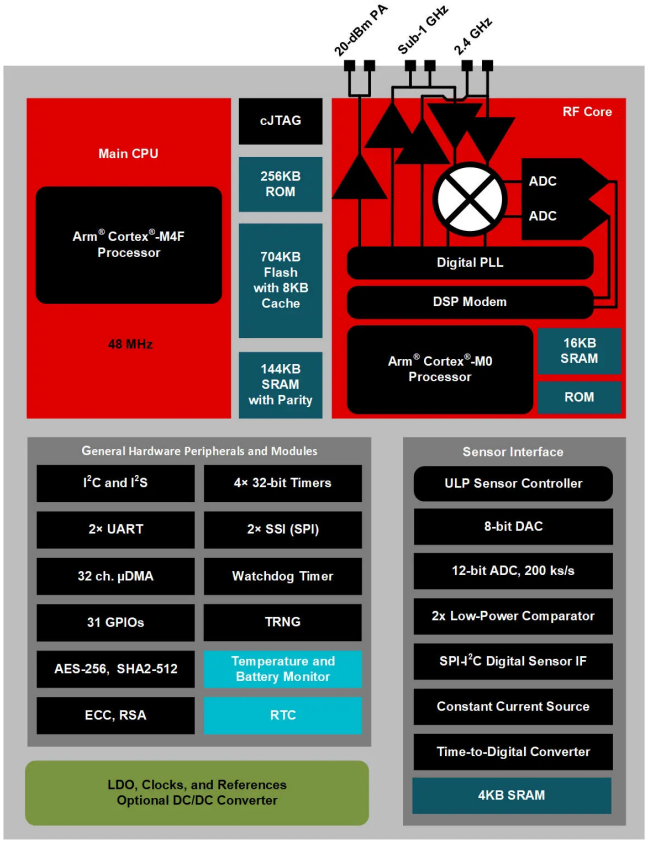

Figure 2: Block diagram of the Texas Instruments CC1352, which include a Cortex-M4F and a Cortex-M0.

2.1 Forces

The increasing capabilities of these platforms make it feasible to use familiar cloud-native workflows for developing applications running on Cortex-A cores. For example, Linux containers can package applications and their dependencies, and continuous integration/continuous deployment (CI/CD) pipelines streamline the development process. Integrating the deployment of software running on Cortex-M cores with the overall application deployment would simplify management. A unified approach allows for the development, deployment, and testing of all software components in unison, enhancing efficiency and coherence across the system.

However, deploying to Cortex-M cores presents several challenges. These cores often operate under stringent resource constraints, with limited processing power and memory compared to Cortex-A cores. Additionally, the development tools and environments for Cortex-M cores can differ significantly from those used in cloud-native workflows, leading to compatibility issues and increased complexity in integration. Portability among different Cortex-M cores is particularly challenging; bare-metal applications often fail to run the same binary across different M versions, and even within the same version due to board-level differences such as I/O configurations. Another significant issue is programmability and discoverability — knowing how to send new workloads to these cores and identifying which cores are present and available for programming. This landscape is fundamentally different from cloud and near-edge environments, where the relative homogeneity of Cortex-A cores, along with the pervasiveness of modern OS's like Linux and sophisticated standard Hardware Abstraction Layers (HAL), significantly lowers the portability barrier. Addressing these challenges requires new approaches to seamlessly integrate Cortex-M deployments into the broader application ecosystem.

Amongst other factors, these considerations we have been discussing present the forces of the patterns. The effectiveness of a pattern language depends not only on the ability of the user to identify themselves with the described problem(s), but also on their ability to carefully balance these forces that will shape the solution. By recognizing and manipulating them, designers can choose the specific trade-offs and tensions that are most representative of their specific problem, and implement the patterns (or slight variants) effectively. Here's non-exhaustive set of forces that shape design decisions for Hybrid Edge systems:

- **Complexity.** The difficulty in managing multiple components, interfaces, and dependencies within the system;
- **Efficiency.** The need to minimize power consumption and optimize resource utilization;
- **Resources.** The need to use/allocate resources (e.g., processing power, memory) effectively among various components and tasks;
- **Portability.** The challenge of ensuring compatibility across different hardware platforms, operating systems, and software environments;
- **Programmability.** The capability in making it easy for developers to send code to be executed on multiple components;
- **Discoverability.** The capability of knowing what components (and their details) are available in a specific system;
- **Development tooling.** The challenge of choosing and integrating development tools that support different frameworks, and hardware platforms, as well as the capability of debugging them in tandem;
- **Security** The challenge of ensuring the security and integrity of data and applications across different components;

2.2 How to read these patterns

A general overview of the interconnection between patterns can be observed in Figure 3. We divide them into four different categories:

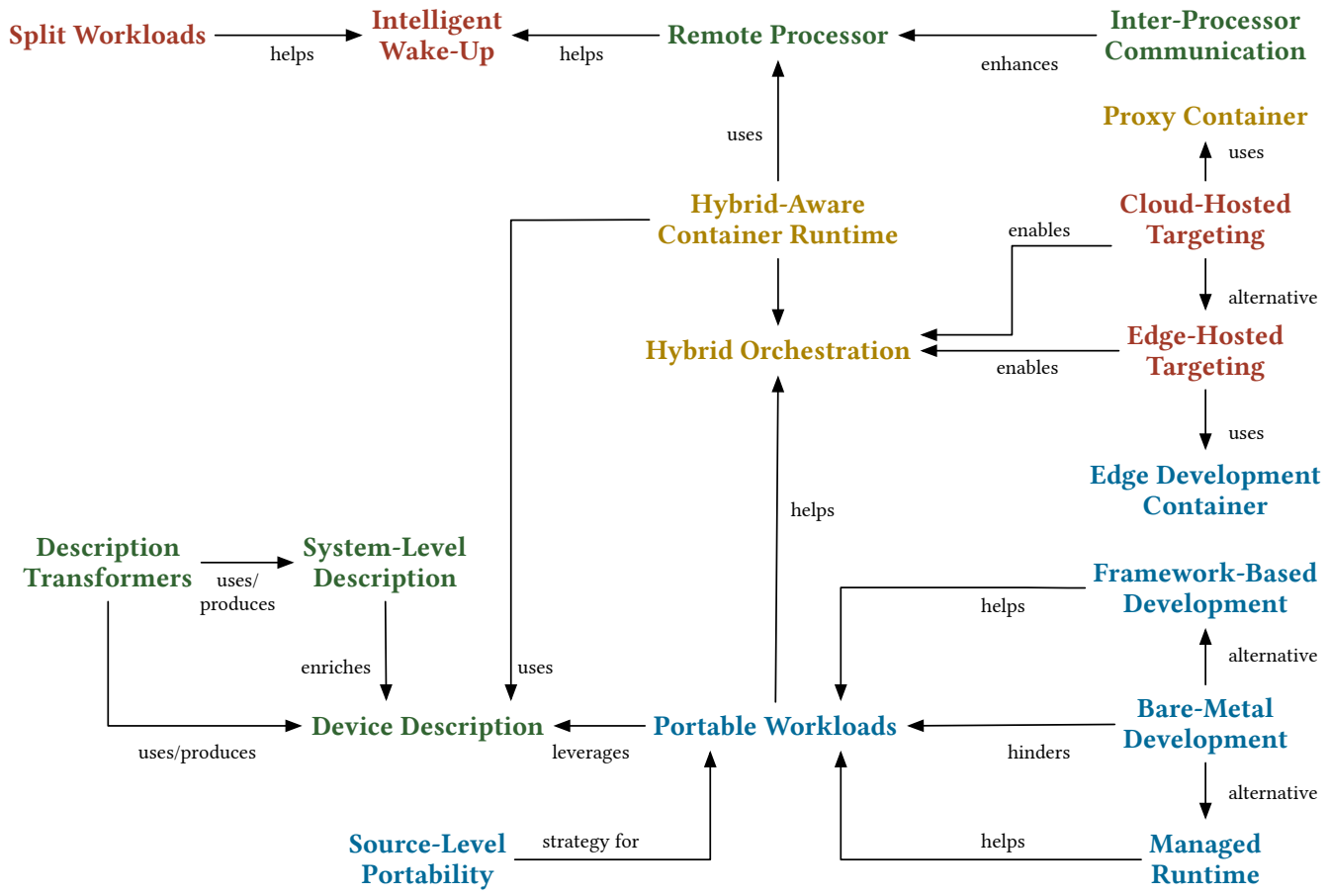

Figure 3: The patterns and their most relevant relationships. Different categories are typeset in different colors, *viz*: (●) Development and Portability, (●) Hardware Abstraction and System Description, (●) Workload Management and Optimization, and (●) Orchestration and System Integration. This figure is color-coded to facilitate visualizing the pattern's categories. Section 2.2 has a textual description of this same information.

(●) **Development and Portability** centers on adaptable coding and deployment practices, making sure applications can be efficiently developed and ported across systems. Patterns in this category are: FRAMEWORK-BASED DEVELOPMENT (3.1), BAREMETAL DEVELOPMENT (3.2), MANAGED RUNTIMES (3.3), SOURCE-LEVEL PORTABILITY (3.10), EDGE DEVELOPMENT CONTAINER (3.16), and PORTABLE WORKLOADS (3.9);

(●) **Hardware Abstraction and System Description** provides methods for managing diverse hardware through abstraction, creating a standardized way to interact with different components. Patterns in this category are: DEVICE DESCRIPTION (3.4), DESCRIPTION TRANSFORMERS (3.5), SYSTEM-LEVEL DESCRIPTION (3.6), REMOTE PROCESSOR (3.7), and INTER-PROCESSOR COMMUNICATION (3.8);

(●) **Workload Management and Optimization** focuses on deploying and optimizing workloads to ensure efficiency, performance, and adaptability in resource-constrained environments. Patterns in this category are: SPLIT WORKLOADS (3.13), INTELLIGENT WAKE-UP (3.14), EDGE-HOSTED TARGETING (3.11), and CLOUD-HOSTED TARGETING (3.12);

(●) **Orchestration and System Integration** handles the coordination of all system components, treating each as part of a cohesive, orchestrated environment. Patterns in this category are: PROXY CONTAINERS (3.15), HYBRID ORCHESTRATION (3.17), and HYBRID-AWARE CONTAINER RUNTIME (3.18).

3 A PATTERN LANGUAGE

This section presents a catalog of patterns designed for deploying workloads on hybrid edge systems. We have identified 18 patterns that form an initial foundation for this pattern catalog. These patterns are derived both from literature review as well as our practical experience in developing hybrid edge systems. Unlike the common style popularized by the GoF book [11], we adopt a more condensed *patlet* style as observed in previous papers [9]. The interaction between patterns (which form this proto-language) can be seen in Figure 3 and a general overview is given on Table 1.

Pattern	Problem	Forces	Solution
Development and Portability (🔵)			
FRAMEWORK-BASED DEVELOPMENT (3.1)	How can we develop applications that work across diverse hardware platforms with minimal modification?	Portability, Cost, Time-to-Market	Use frameworks that provide hardware abstraction layers and built-in libraries to standardize development across platforms.
BAREMETAL DEVELOPMENT (3.2)	How can we maximize performance by fully exploiting hardware capabilities?	Performance, Hardware Control, Complexity	Target the hardware without abstractions, optimizing for specific capabilities.
MANAGED RUNTIMES (3.3)	How can we simplify the management of multiple workloads with rapid development cycles?	Maintainability, Development Speed, Flexibility	Utilize managed runtimes, like microPython or WASM, to streamline development and enable dynamic workload management.
PORTABLE WORKLOADS (3.9)	How can we develop workloads that run seamlessly across different architectures?	Portability, Flexibility, Abstraction	Develop cross-platform compatible workloads using standardized APIs and frameworks.
SOURCE-LEVEL PORTABILITY (3.10)	How can we adapt software to multiple platforms without needing binary-level compatibility?	Portability, Abstraction, Maintainability	Retain source code until final compilation for specific platforms, optimizing it for each target's unique features.
EDGE DEVELOPMENT CONTAINER (3.16)	How can we standardize development environments for hybrid systems?	Consistency, Portability, Efficiency	Use containerized development environments to encapsulate tools and dependencies for consistent deployment.
Hardware Abstraction and System Description (🟢)			
DEVICE DESCRIPTION (3.4)	How can we achieve application portability across boards with differing hardware configurations?	Portability, Abstraction, Complexity	Adopt device descriptions that standardize hardware interfaces, allowing software to be adaptable across platforms.
DESCRIPTION TRANSFORMERS (3.5)	How can we ensure device descriptions are compatible across different operating systems?	Interoperability, Abstraction, Flexibility	Use transformers to convert device descriptions into formats compatible with different OS requirements.
SYSTEM-LEVEL DESCRIPTION (3.6)	How can we provide a unified view of all hardware components within a system?	System Complexity, Efficiency, Flexibility	Implement system-level descriptions that detail all system components and their interactions.
REMOTE PROCESSOR (3.7)	How can we manage diverse processing units within a hybrid system?	Scalability, System Coordination, Abstraction	Use a framework that abstracts and standardizse access to heterogeneous computing units.
INTER-PROCESSOR COMMUNICATION (3.8)	How can we facilitate efficient communication between heterogeneous processing units?	Synchronization, Portability, Performance	Adopt an inter-processor communication framework to provide a uniform interface for data exchange.
Workload Management and Optimization (🔴)			
EDGE-HOSTED TARGETING (3.11)	How can we adapt applications in real-time to specific edge devices?	Adaptability, Resource Utilization, Real-time Efficiency	Perform last-stage compilation on or near the edge device, optimizing based on real-time hardware conditions.
CLOUD-HOSTED TARGETING (3.12)	How can we conserve device resources while ensuring compilation consistency?	Resource Efficiency, Consistency, Security	Offload compilation to cloud servers, ensuring centralized control over the build environment.
SPLIT WORKLOADS (3.13)	How can we optimize resource use by distributing tasks across different processing units?	Efficiency, Performance, Scalability	Divide the application into modules that leverage both high-power and low-power processors for respective tasks.
INTELLIGENT WAKE-UP (3.14)	How can we reduce energy consumption in hybrid systems?	Energy Efficiency, Responsiveness, Complexity	Suspend subsystems during low-demand periods and use wake-up mechanisms for resuming tasks as needed.
Orchestration and System Integration (🟡)			
PROXY CONTAINERS (3.15)	How can we manage microcontrollers using container orchestration tools?	Orchestration, Abstraction, Flexibility	Deploy a proxy container on the main processor to handle workload management for MCUs.
HYBRID ORCHESTRATION (3.17)	How can we manage diverse computational resources uniformly within an orchestration framework?	Resource Allocation, Efficiency, Flexibility	Treat all units, including microcontrollers, as first-class entities within the orchestration framework.
HYBRID-AWARE CONTAINER RUNTIME (3.18)	How can we adapt container runtimes to handle diverse hardware configurations?	Scalability, System Integration, Flexibility	Extend container runtimes to recognize and manage diverse computational units within hybrid systems.

Table 1: General overview of the patterns and their problems, main forces, and solutions, grouped by category.

3.1 Framework-Based Development (●)

You are developing for a hybrid system and aim to deploy parts of your application on the MCU. One approach is to develop a custom-tailored workload using baremetal development. However, different boards (and even different revisions) have varying capabilities and designs. Pin layout may be different, and the connected devices might have slight differences. Additionally, different MCU families often possess different capabilities when interacting with peripherals. Performance is not your main priority; rather, you need to target a wide variety of boards, and aim for a lower time-to-market (TTM) and total cost of ownership (TCO). *How can one develop applications that work across diverse hardware platforms with minimal modification?*

Therefore, use an established framework that abstracts most of the low-level variability between different boards. Frameworks such as the Arduino framework, RTOS, and Mbed OS provide several functionalities out of the box, including potential hardware abstraction layers. These frameworks simplify the development process by offering ready-made solutions for common tasks, improving portability across different boards, and reducing the cost of development.

Rationale. Framework-based development leverages existing tools and libraries, allowing for faster deployment and reduced costs. The hardware abstraction layers provided by these frameworks help mitigate the differences between various boards and MCUs, enhancing portability. Although this approach may not offer the highest performance, it significantly reduces development time and costs, making it ideal for projects where flexibility and cost efficiency are paramount. Using established frameworks also ensures a more manageable and maintainable codebase, contributing to overall system robustness and scalability. **Also see:** BAREMETAL DEVELOPMENT (3.2), PORTABLE WORKLOADS (3.9).

3.2 Baremetal Development (●)

You are developing for a hybrid system and you need to deploy parts of your application on the MCU. One approach is to use framework-based development to simplify the process. However, this may not always meet your performance requirements. When targeting specific hardware capabilities and optimising for performance is critical, you might feel that the frameworks are introducing overheads, limitations, and non-transparent abstractions. *How can one maximize performance by fully exploiting hardware capabilities?*

Therefore, develop for your target minimising the usage of frameworks. This approach involves writing code that directly interacts with the hardware without relying on abstractions. By doing so, you can fully exploit all the hardware's capabilities and achieve the highest possible performance.

Rationale. Baremetal workload development provides maximum control over the hardware, allowing you to fine-tune performance and utilize specific features of the MCU. This approach eliminates the overhead introduced by frameworks, resulting in more efficient use of resources. Although it requires a deeper understanding of the hardware and more development effort, the trade-off is justified when performance and precise control are paramount. Also, frameworks might not always support the hardware you intend to target, and extending them might incur into a maintainability cost. Baremetal development is ideal for applications where latency, resource utilization, and real-time performance are critical. It also allows for more customized and optimized solutions tailored to specific hardware configurations, making it a preferred choice for high-performance and mission-critical systems where the hardware is not expected to change or evolve, and TTO and TTM are not the main factors at play. **Also see:** FRAMEWORK-BASED DEVELOPMENT (3.1), MANAGED RUNTIMES (3.3), PORTABLE WORKLOADS (3.9).

3.3 Managed Runtimes (●)

You are developing for a hybrid system and you feel the complexity of managing multiple workloads and the need for quick development cycles present significant challenges. Writing low-level code can be time-consuming and difficult to maintain, especially when rapid iteration and deployment are required. Furthermore, the interaction between the application running on the A and the one running on the M is cumbersome, and you feel the need for something for homogeneous and maintainable. You might also have the need for some dynamism in your system (i.e., managing multiple tasks in runtime). *How can one simplify the management of multiple workloads with rapid development cycles?*

Therefore, use a managed runtime, like microPython or WASM. These runtimes facilitate faster development times and easier management of multiple workloads. Managed runtimes provide built-in tools and libraries, making it simpler to add abstractions and streamline development processes.

Rationale. Managed runtimes significantly accelerate the development process by offering high-level programming environments that abstract away many low-level details as well as providing an easy path for the addition of new features and abstractions. Achieving complex functionality (like managing multiple workloads) can also be tackled by incorporated such functionalities in the runtime. However, the increased memory usage and typically lower performance compared to baremetal development and even framework-based development are notable drawbacks particularly for performance-critical applications. In relationship to the latter, managed runtimes do offer strong dynamic support, often being relatively trivial to change workloads without needing to completely reprogram the MCU. As thus, managed runtimes are advantageous in scenarios where rapid development, flexibility, and ease of maintenance are more critical than achieving the highest possible performance. There are multiple environments suitable to be executed in MCUs like microPython, JavaScript, WASM, and Lua. **Also see:** BAREMETAL DEVELOPMENT (3.2), PORTABLE WORKLOADS (3.9).

3.4 Device Description (●)

You are developing for an hybrid system, and you want to achieve portability of your applications. For this, it is crucial to know how to communicate with your MCU, as well as the connected peripherals. Depending on the board, the same peripheral can be connected using the same protocol (e.g., I2C), but using different pins. Sometimes, it can even use different protocols. Same applies for communication with your MCU. Memory mapping is sometimes used, but this too can vary across different boards.

Therefore, adopt some sort of descriptor that provides the necessary information to allow multiple targets. Use this descriptor either during the execution of the application, by inspecting the information, or during compilation, by generating tailored code.

Rationale. A device description facilitates portability by standardizing the way computing units, interconnects and peripherals are described and accessed, regardless of the underlying hardware variations. Examples of such device descriptions include the "device tree," used by Linux and Zephyr. Linux provides the device tree in a way that can be inspected by the application, while Zephyr uses the device tree to generate headers so that the application can be compiled for the intended target. By providing a structured method to define hardware configurations, device descriptions allow applications to be more adaptable and reduce the need for hardware-specific code. This abstraction mechanism makes it easier to develop software that can run on multiple platforms without extensive modification. The disadvantages of relying on a device description is that it might increase the application complexity, require additional maintaining efforts (i.e. up-to-date descriptions), and might add some potential performance overheads. **Also see:** DEVICE DESCRIPTION (3.4), SYSTEM-LEVEL DESCRIPTION (3.6), PORTABLE WORKLOADS (3.9).

3.5 Description Transformers (●)

In hybrid systems, achieving portability often involves using device descriptions to enable seamless interaction with connected peripherals and other computing units. However, not all operating systems use device descriptions in the same way. For instance, although Linux and Zephyr both use device trees, their implementations are not compatible.

Therefore, rely on a single source of truth, and apply transformations to achieve the intended target. This approach leverages transformers of descriptors to make them usable across multiple operating systems.

Rationale. Description transformation enables interoperability between different operating systems by standardising device descriptions and converting them into formats that each OS can understand. This approach reduces the complexity of developing for multiple platforms and enhances portability. By transforming device descriptions, developers can write code that is adaptable to various operating systems without needing to rewrite or significantly modify the hardware interaction layer. However, this process introduces additional layers of abstraction and potential performance overhead. It also adds to an increase maintenance effort, as not only the descriptions need to be kept up-to-date, but the transformers as well. An example is the System Device Tree and Lopper [2], the latter providing a tool to transform device descriptions (i.e. device trees) to be compatible with different target environments. Although Lopper [2] supports multiple basic transformations out-of-the-box, most of the useful ones need to be written in Python. **Also see:** SYSTEM-LEVEL DESCRIPTION (3.6), DEVICE DESCRIPTION (3.4).

3.6 System-Level Description (●)

Modern computing systems are comprised of complex architectures that include not just the CPUs and MCUs, but also multiple clock and power domains, peripheral devices, and their interconnections. Some systems might even contain dynamic units such as FPGA's, where "soft" cores might provide further functionality to the system. In order for Operating Systems and tools to leverage all available resources, it is crucial to have a comprehensive description that encompasses all components of the system. This detailed description is necessary for tasks such as booting, configuring, and optimising the system, as well as managing and deploying payloads to each different computing unit. The typical system device tree might fall short in providing the necessary detail and flexibility for these cases, as it typically provides a view centred around a single computing unit (i.e. the CPU). We need to accurately describe all elements and their interactions within a system or board, ensuring that software can efficiently manage and utilize them.

Therefore, use a system-level description that details (in a meaningful way) every system component and their interconnections.

Rationale. A system-level description provides a unified and detailed representation of all components and their relationships within a system. It describes multiple CPUs, MCUs, clock domains, and power domains — amongst other details — ensuring that most (if not all) aspects of the system are considered. This facilitates better management, configuration, and optimisation of the available resources by software, enabling a more efficient and robust development lifecycle. This level of detail is particularly needed in complex systems the control and orchestration of various components are desired. Despite the benefits of improved system representation, creating and maintaining such detailed descriptions can be challenging, though. An example of this pattern is the System Device Tree (SDT) which provides a hierarchical and comprehensive view of the system's hardware, but with little semantics attached. Instead, a transforming tool called Lopper [2] is used to regard the SDT as a single source of truth, and then manipulate it in such a way that is further usable by different OS's (such as Linux and Zephyr) with the appropriate PoV. Another example is the use of system overlays to enhance the current description with system-level details. **Also see:** DESCRIPTION TRANSFORMERS (3.5), DEVICE DESCRIPTION (3.4).

3.7 Remote Processor (●)

Modern SoCs typically have heterogeneous remote processor devices in Asymmetric MultiProcessing (AMP) configurations, running different instances of operating systems, such as Linux or various real-time OS flavours. For example, the BeaglePlay board uses a TI Sitara AM625 SoC with 1.4GHz quad-core Arm Cortex-A53, two M4F microcontrollers (one in the same SoC, and additional one in the board), and a PRU. Typically, the quad-core Cortex-A53 runs Linux in an SMP configuration, while the other cores run their own baremetal or RTOS instances. Managing these remote processors can be challenging due to their diverse architectures and configurations.

Therefore, abstract the access to computing units by using a remote processor framework. This allows different platforms and architectures to control attached computing units while abstracting the hardware differences.

Rationale. The remote processor framework provides a unified method to manage heterogeneous computing units within a hybrid system. By abstracting hardware differences, it simplifies the development and maintenance of drivers, ensuring that remote processors can be controlled and communicated with efficiently. This abstraction layer makes it easier to develop software that can run on multiple platforms without extensive modification. An example of such a framework is `remoteproc` [4], which standardises the interaction with attached (companion?) processors. However, relying on a remote processor framework introduces trade-offs, such as increased dependency on the framework and incompatibility with several available boards due to strict programming models. Interestingly, the overheads introduced by a framework such as `remoteproc` are relatively low, so performance is not usually a problem. The main benefits are a streamlined development environment, and improved portability across different platforms. **Also see:** INTER-PROCESSOR COMMUNICATION (3.8), INTELLIGENT WAKE-UP (3.14), HYBRID-AWARE CONTAINER RUNTIME (3.18).

3.8 Inter-Processor Communication (●)

In modern computing systems with heterogeneous architectures, multiple components such as CPUs, MCUs, Digital Signal Processors (DSPs), Field-programmable Gate Arrays (FPGAs), and other accelerators need to collaborate efficiently. Enabling efficient, reliable, and scalable communication between the different computing units of such system is not trivial. These units might have varying capabilities, architectures, and communication protocols, integrated in specific ways by the board designer. Directly implementing communication for each pair of units can lead to complex, error-prone, and non-portable code. Furthermore, handling issues like synchronisation, integrity, and latency is challenging when units may operate in different clock domains, with different priorities.

Therefore, abstract the details of the communications in a framework, and provide an uniform interface for all the computing units to pass messages between them.

Rationale. Adopting a framework simplifies the development and maintenance of inter-processor communication by abstracting low-level details and providing a uniform interface. It enhances portability, scalability, and reliability while optimising performance and resource management. The use of such a framework ensures that communication between heterogeneous computing units is efficient and reliable, even as systems' complexity grows. Developers can ensure seamless coordination and data sharing among diverse computing units. However, introducing an abstraction layer may add some overheads. The Remote Processor Messaging [5] is an example of such IPC framework, that is designed to work with REMOTE PROCESSOR (3.7).

3.9 Portable Workloads (●)

You are developing for an hybrid system composed of multiple boards with varying architectural capabilities. You want to orchestrate these systems in a frictionless, cloud-native manner, without worrying too much about the particular details of each board; rather, you would prefer your application to run seamlessly across all devices that comprise your system. You are already using containerisation techniques to abstract the underlying hardware differences, and this works well for applications targeting full-blown operating systems like Linux. But you want to leverage all computation power available, including microprocessors. For example, you might have decided that you want to split your workloads and use intelligent wake-up. Differences in processor types, memory architectures, and peripheral interfaces pose significant challenges to achieving portability.

Therefore, aim to develop workloads with cross-platform compatibility in mind, leveraging standardised APIs and frameworks, or other types of middleware that can manage and expose resources and hide the complexities of targeting heterogeneous devices.

Rationale. Portable workloads enable flexible deployment, allowing the same workload to run on different devices without modification. This facilitates scalability by adding new devices and ensures efficient resource utilisation by dynamically allocating processing power, memory, and storage. It also simplifies maintenance and updates, by decoupling the workloads from the specific hardware they run on. The usage of portable programming languages, frameworks and runtimes can ensure wider compatibility across different platforms. Prioritising portability, though, may introduce significant challenges and overheads, such as (a) managing hardware-specific resources, (b) trade-offs between abstraction layers and optimisations, and (c) ensuring consistency in terms of execution performance requirements. Technologies like Docker and Kubernetes already allow you to encapsulate workloads, making them somewhat independent of the underlying hardware, but they mostly assume that you are running Linux, which would not be the case if you are targeting Hybrid systems and you have split your workloads to leverage multiple processors. **Also see:** SOURCE-LEVEL PORTABILITY (3.10), MANAGED RUNTIMES (3.3), FRAMEWORK-BASED DEVELOPMENT (3.1), BAREMETAL DEVELOPMENT (3.2), DEVICE DESCRIPTION (3.4).

3.10 Source-Level Portability (●)

You are developing for a hybrid system composed of multiple boards with varying architectural capabilities. Achieving binary-level portability, especially when targeting MCUs, is challenging due to differences in processor types, memory architectures, and peripheral interfaces. You want to orchestrate these systems in a frictionless, cloud-native manner, ensuring that your application can run seamlessly across all devices within your system. You need Portable Workloads, but you want to avoid the overheads introduced by Managed Runtimes, while still coping with the various differences in your system.

Therefore, adopt source-level portability. This approach involves preserving the source code (or some form of intermediate or object representation) until it is targeted for a new SoC/board. At that point, a final stage of compilation takes into account the specific details of the target system, including available resources and I/O specificities.

Rationale. Source-level portability enables portable and flexible deployment by allowing the same source code to be compiled for different targets without manual intervention. This approach leverages last-stage compilation techniques to tailor the code for specific hardware, ensuring correct and optimal use of available resources by accommodating system-specific details. It simplifies the maintenance and updates by keeping the core codebase consistent while adapting the compiled output to the various platforms present in a system. It might also provide increased benefits by toggling specific optimisation techniques tailored for the desired target. However, this pattern introduces challenges in maintaining source-level compatibility and managing system-specific configurations across your system. In some scenarios, full availability of source-code might also be impossible (such as when using third-party libraries), or even pose a risk regarding intellectual property leakage. **Also see:** PORTABLE WORKLOADS (3.9), EDGE-HOSTED TARGETING (3.11), and CLOUD-HOSTED TARGETING (3.12).

3.11 Edge-hosted Targeting (●)

You are developing for a hybrid system composed of multiple boards with varying architectural capabilities. Achieving source-level portability is crucial for ensuring that your application can run seamlessly across all devices within your system. You want to prioritise device adaptation and characterisation, and reduce deployment time, even in scenarios where connectivity may be erratic.

Therefore, perform the last-state compilation of your workloads near the target, leveraging the available hardware resources to compile the code specifically for that device.

Rationale. On-device compilation provides significant advantages in terms of real-time adaptation and reduced deployment time, by taking into account the exact state of the hardware at the time of deployment and optimising the compilation for specific conditions. By compiling directly on or near the target device, the system can make better choices based on the local system description, which is particularly useful in dynamic environments where hardware components might change or be upgraded. However, this approach can be resource-intensive, consuming CPU, memory, and energy, which might be scarce on the target device, especially on low-power boards. A common mitigation approach is to offload the compilation to near-edge devices, such as gateways, if available. Additionally, implementing a reliable on-device compilation process adds complexity to the system, requiring robust error handling, security considerations, and additional software dependencies on the target device. Ensuring all necessary tools and libraries for compilation are available on the target device can also be challenging, particularly in constrained environments. **Also see:** CLOUD-HOSTED TARGETING (3.12), EDGE DEVELOPMENT CONTAINER (3.16), HYBRID ORCHESTRATION (3.17).

3.12 Cloud-hosted Targeting (●)

You are developing for a hybrid system composed of multiple boards with varying architectural capabilities. You have decided to achieve source-level portability, as you find it essential for ensuring that your application runs seamlessly across all devices within your system. However, you want to make sure your devices are being used in the most mission and energy-efficient manner and minimise the risk regarding intellectual property leakage.

Therefore, perform the last-stage compilation centrally in the cloud, leveraging typical CI/CD servers to handle the build process. This allows for centralised management of the compilation environment, ensuring consistency and higher control over dependencies and source-code.

Rationale. Cloud-hosted targeting conserves resources on target devices by offloading the compilation process to powerful CI/CD servers, allowing target devices to focus on running workloads with any further "distractions". Centralised compilation ensures consistency and control over the build environment, enhancing security by reducing the need to install and maintain tools and compilers on the target device. This approach also simplifies the development workflow, enabling automation of the build, test, and deployment processes. However, pre-deployment compilation may not account for real-time changes in hardware configurations or resource availability on the target device, potentially leading to less optimised code. This can be mitigated by maintaining an accurate and up-to-date system description that encapsulates all target-specific details; but such techniques also comes with added overheads and is error-prone; usually, it ends up in a combinatorial explosion of pre-emptively built binaries for each possible combination (some of which might never be used). Despite these challenges, the additional computational power available can be leveraged to execute fully automated CI/CD pipelines that are integrated with version control, testing frameworks, and deployment tools to provide a streamlined workflow, freeing the constrained resources for what they are most needed to. **Also see:** PROXY CONTAINERS (3.15), EDGE-HOSTED TARGETING (3.11), HYBRID ORCHESTRATION (3.17).

3.13 Split Workloads (●)

Your are developing for an hybrid systems that combines a powerful application processor (A), and energy-efficient microcontrollers (M). You want to maximize the resource utilisation and efficiency. For example, in a smart camera performing object recognition, you want the application to leverage the computing power of the A for intensive tasks. But that consumes power. Simultaneously, you have an energy-efficient unit in your system that, albeit limited, could potentially perform a lightweight object detection and trigger the power-hungry features only when necessary. Directly running the entire workload on a single processor may lead to inefficient use of resources, higher power consumption.

Therefore, divide your application into different modules that can run on both the A and the M in a collaborative fashion. Lightweight tasks are handled by the M, while the A takes on more computationally intensive tasks.

Rationale. Splitting workloads into complementary and collaborating modules enables efficient utilisation of both high-power and low-power processors. In the example above, by assigning lightweight, low-framerate tasks to the M, and reserving the A for intensive tasks, the overall system efficiency improves. This approach reduces power consumption, as the A can remain in a low-power state until needed, and enhances performance by distributing tasks

according to processor capabilities. Efficiently distributing the application workload is a challenging task though, that adds complexity to the application design, and requires increased coordination between multiple parts. Incidentally, it might also provide increased resiliency, as (at least some) of the application functions might gain some level of redundancy. This pattern is especially useful in systems where continuous operation is required but high computational power is only intermittently needed. **Also see:** INTELLIGENT WAKE-UP (3.14).

3.14 Intelligent Wake-Up (●)

You are developing for an hybrid systems where energy efficiency is a priority. An appropriate management of power consumption can significantly extend battery life and reduce operational costs. Your application does not require the full capabilities of your system to be on all the time. Instead, you have a means to balance some trade-offs by offloading parts of your application that rely singly on specialised computing units by Splitting Workloads into independent units.

Therefore, suspend unnecessary subsystems, by selectively identifying parts of the system that are not required for immediate tasks, and then placing state preservation techniques and wake-up mechanisms to resume operations seamlessly when required.

Rationale. Suspending subsystems helps in achieving energy efficiency by reducing power consumption during idle or low-demand periods. By leveraging low-energy cores for lightweight tasks and suspending high-power subsystems, the overall energy footprint of the system can be minimised. For example, running a workload on a low-energy core like an MCU while suspending the main processor and entering a low-power state can achieve substantial energy savings. This usually also presents a kind of cascading effect: the CPU typically runs from DRAM, whereas an MCU might be executing code from SRAM; this means there's an opportunity to leverage compounding gains from efficiency. The challenge is on identifying and suspending subsystems that are not actively needed without compromising the system's functionality or performance, provided the software (*cf.* SPLIT WORKLOADS (3.13)) and the system (*cf.* REMOTE PROCESSOR (3.7)) was designed to allow such features. This involves coordinating the suspension and wake-up processes, preserving subsystem states, and ensuring quick transitions to maintain system responsiveness.

3.15 Proxy Containers (●)

You are developing for an hybrid systems and you want to orchestrate workloads across multiple computational units containing both application processors (A) and energy-efficient microcontrollers (M). Current orchestration frameworks like Kubernetes often lack both the visibility and the capability to treat these diverse units as first-class objects, complicating the management of workloads across multiple SoCs/Boards.

Therefore, deploy a container on the application processor (A) that contains all the necessary tools and capabilities to manage the microcontroller (M). The proxy container acts as an intermediary, handling the deployment, execution, and management of workloads on the M within an isolated environment.

Rationale. Proxy containers provide a simple and practical solution for managing heterogeneous systems by leveraging the capabilities of the A to orchestrate the M. This approach isolates the management tasks within a dedicated container, ensuring that the orchestrator can handle diverse workloads without the need for native support for each type of computational unit. The proxy container typically include tools for communication, monitoring, and control, enabling the deployment and execution of workloads on the M. Although this introduces additional complexity in managing the proxy container itself (and setting it up to have the necessary access to the required hardware), it is a much simpler alternative to full-blown hybrid orchestration, providing a subset of its benefits such as simplified management. Proxy containers are a straightforward mechanism to allow existing orchestration frameworks to manage hybrid systems, enhancing the flexibility and scalability of the deployment. **Also see:** CLOUD-HOSTED TARGETING (3.12).

3.16 Edge Development Container (●)

You are developing for an hybrid and heterogeneous systems and you need portable workloads. You have decided to use source-level portability, which involves compiling and preparing applications to run on various target devices. This process requires consistent and reliable development environments, equipped with all necessary tools and dependencies. For example, targeting the NXP i.MX8 might require the ARM GCC, specific SDKs for the specific MCU family, and additional libraries and configurations, some of which may be provided by the vendor. Ensuring that these environments are easily replicatable and adaptable to different target systems is crucial as slight variations in development environments can lead to inconsistencies and deployment issues, complicating the development process.

Therefore, create containerised development environments that include all the necessary tools and dependencies to target specific systems. These containers encapsulate the entire development setup, ensuring consistency and portability across different development and deployment platforms.

Rationale. Development containers provide a standardised, reproducible development environment that simplifies the process of targeting multiple systems. By encapsulating compilers, SDKs, libraries, and multiple other tools within a single container, developers and system integrators can ensure that the same environment is used across different stages of the lifecycle, ranging from local development to CI/CD pipelines. This approach enhances consistency, reduces setup time, and minimizes the risk of environment-related issues by enabling developers to focus on coding and testing, rather than managing complex toolchains and dependencies. It can also be extremely helpful for achieving hybrid orchestration, irrespective of a cloud-hosted or edge-hosted targeting strategy being adopted. For example, in edge-hosted targeting, development containers can be built, cached, and sent to the edge device with the minimal environment necessary for it to perform a last stage-compilation, and then clean-up afterwards. However, creating and maintaining development containers do require some initial effort and infrastructure; this trade-off must be balanced with the long-term benefits of streamlined development processes, improved portability, and

reduced deployment issues to make it a valuable practice. **Also see:** EDGE-HOSTED TARGETING (3.11).

3.17 Hybrid Orchestration (●)

In hybrid and heterogeneous systems, orchestrating multiple computing units presents significant challenges. These systems often include a mix of powerful CPUs, energy-efficient MCUs, and various peripheral devices, each with different capabilities and requirements. For example, in a smart city application, various devices such as sensors, cameras, and controllers need to work together seamlessly, with powerful servers, edge devices, and MCUs as integral parts of the system, each running their different workloads based on their capabilities. Traditional orchestration frameworks often overlook smaller computing units like MCUs, treating them as secondary components rather than integral parts of the system. The goal is to manage these diverse resources in a frictionless, uniform manner, ensuring seamless operation and optimal utilisation.

Therefore, treat all computational resources, including MCUs, as first-class units within the orchestration framework. By doing so, the framework can manage, schedule, and optimise workloads across the entire spectrum of devices, ensuring efficient use of all available resources.

Rationale. Hybrid orchestration provides a unified management interface for diverse computing units, simplifying the development, deployment, and maintenance of applications in heterogeneous environments. By considering MCUs and other low-power devices as first-class units, the orchestration framework can operate in a typical cloud-native fashion, optimising workload distribution based on the specific capabilities and constraints of each SoC/Board. For instance, lightweight tasks like data collection from sensors can be handled by MCUs, while more computationally intensive tasks like data analysis and pattern recognition are offloaded to powerful edge devices or cloud servers. This leads to better resource allocation, improved energy efficiency, reduced latency, and enhanced overall performance and responsiveness of the system as a whole. However, achieving hybrid orchestration introduces non-negligible complexity in terms of integrating all of the diverse hardware into a cohesive management framework. **Also see:** PORTABLE WORKLOADS (3.9), CLOUD-HOSTED TARGETING (3.12), EDGE-HOSTED TARGETING (3.11), HYBRID-AWARE CONTAINER RUNTIME (3.18).

3.18 Hybrid-Aware Container Runtime (●)

You are developing for an hybrid and heterogeneous systems, and you need to orchestrate multiple computing units. You have splited your workloads, with lightweight data collection tasks running on MCUs, while data processing and analytics tasks are being handled by more powerful CPUs. But each workload has different requirements related to performance, connected devices (such as cameras), etc. Traditional container runtimes primarily designed for homogeneous environments do not possess the capabilities to handle diverse hardware configurations, including MCUs and other computational units present in a single SoC/Board. Enabling existing orchestration frameworks (*e.g.* Kubernetes) to manage and optimize workloads across a variety of computational units within a hybrid system involves integrating low-power devices and specialised processors into the orchestration framework, ensuring they are treated as first-class units.

Therefore, reuse existing frameworks by extending the capabilities of the container runtime to recognise, manage and expose different computational units in tandem with the orchestrator.

Rationale. A hybrid-aware container runtime allows for the seamless integration of various computational units into existing orchestration frameworks. By enhancing the container runtime to work with diverse hardware, it ensures that workloads can be efficiently distributed and managed at scale based on the specific capabilities and constraints of each SoC/Board. It also simplifies the development and deployment processes by providing a unified, robust, cloud-like management interface for hybrid and heterogeneous systems. These kind of runtimes can also leverage system descriptors to understand the hardware capabilities and constraints of each device, and use remote processor frameworks to coordinate the deployment and execution of workloads. However, given the lack of widely available orchestrators that are natively hybrid-aware, extending the container runtime does introduce additional complexity, and requires careful consideration of the trade-off between the benefits of a uniform workload management and added effort. **Also see:** HYBRID ORCHESTRATION (3.17), REMOTE PROCESSOR (3.7), DEVICE DESCRIPTION (3.4).

4 CONCLUSIONS

In this work, we presented a pattern language comprised of eighteen (18) patterns, which provides a framework for navigating the challenges related to the growing importance of cloud-native practices in Hybrid Edge computing. These challenges, and their corresponding solutions, are the result of the interplay between competing forces, such as resource allocation, development tooling, portability, programmability, and discoverability, amongst others. While discussing these patterns, we have also identified key areas where further investigation is needed to fully realise the potential of hybrid systems at the Edge. Specifically, portability remains a major concern, and we advocate for the development of advanced tools that can effectively manage the software lifecycle across multiple hardware platforms and environments. Ultimately, this paper also underscores the need for continued research into hybrid systems, in order to tackle its increasing need for efficient, scalable, and adaptive computing at the Edge.

ACKNOWLEDGMENTS

The authors would like to thank James Noble for helping to review preliminary versions of this work (as our “shepherd”) and to the PLoP Writer’s Workshop members of group #4 for their insightful feedback during the session. We would also like thank Alexandre Ferreira for all the feedback received during our research activity.

REFERENCES

[1] [n.d.]. AWS & Carrier Co-Development — aws.amazon.com. https://aws.amazon.com/campaigns/carrier/. [Accessed 20-05-2024].
[2] [n.d.]. devicetree-org/lopper — github.com. https://github.com/devicetree-org/lopper. [Accessed 15-10-2024].

[3] [n.d.]. FreeRTOS - Market leading RTOS (Real Time Operating System) for embedded systems with Internet of Things extensions — freertos.org. https://www.freertos.org/index.html. [Accessed 20-05-2024].
[4] [n.d.]. Remote Processor Framework; The Linux Kernel documentation. https://docs.kernel.org/staging/remoteproc.html. [Accessed 05-08-2024].
[5] [n.d.]. Remote Processor Messaging (rpmsg) Framework; The Linux Kernel documentation. https://docs.kernel.org/staging/rpmsg.html. [Accessed 06-08-2024].
[6] [n.d.]. The Volkswagen Group on AWS: Case Studies, Videos, Innovator Stories — aws.amazon.com. https://aws.amazon.com/solutions/case-studies/innovators/volkswagen-group/. [Accessed 20-05-2024].
[7] [n.d.]. The Zephyr Project – A proven RTOS ecosystem, by developers, for developers. — zephyrproject.org. https://www.zephyrproject.org. [Accessed 20-05-2024].
[8] [n.d.]. Traeger Grills & OST – Amazon Web Services (AWS) — aws.amazon.com. https://aws.amazon.com/partners/success/traeger-grills-ost/. [Accessed 20-05-2024].
[9] João Pedro Dias, Tiago Boldt Sousa, André Restivo, and Hugo Sereno Ferreira. 2020. A Pattern-Language for Self-Healing Internet-of-Things Systems. In *Proceedings of the European Conference on Pattern Languages of Programs 2020* (Virtual Event, Germany) *(EuroPLoP '20)*. Association for Computing Machinery, New York, NY, USA, Article 25, 17 pages. https://doi.org/10.1145/3424771.3424804
[10] João Pedro Dias, André Restivo, and Hugo Sereno Ferreira. 2022. Designing and constructing internet-of-Things systems: An overview of the ecosystem. *Internet of Things* 19 (2022), 100529. https://doi.org/10.1016/j.iot.2022.100529
[11] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, Boston, MA, USA.
[12] Weisong Shi and Schahram Dustdar. 2016. The Promise of Edge Computing. *Computer* 49, 5 (2016), 78–81. https://doi.org/10.1109/MC.2016.145