Elastic AI: system support for adaptive machine learning in pervasive computing systems

Christopher Cichiwskyj¹ · Stephan Schmeißer¹ · Chao Qian¹ · Lukas Einhaus¹ · Christopher Ringhofer¹ · Gregor Schiele¹

Received: 12 March 2021 / Accepted: 5 June 2021 / Published online: 19 July 2021
© The Author(s) 2021

Abstract

Artificial intelligence (AI) is an important part of today's pervasive computing systems. Still, there is no end-to-end system platform that allows to deploy, update, manage and execute AI models in pervasive systems. We propose such a system platform in this paper. Most importantly, we reuse concepts and techniques from twenty years of pervasive computing research on how to enable runtime adaptation and apply it to AI. This allows to specify adaptive AI models that are able to react to a multitude of dynamic changes, e.g. with respect to available devices, networking conditions, but also application requirements and sensor data sources. Developers can optimise their applications iteratively, starting with a generic setup and refining it step by step towards their specific pervasive computing scenario. To show the applicability of our platform, we apply it to two pervasive use cases and evaluate them, achieving up to four times faster inference and three times lower energy consumption compared to a classical AI deployment.

Keywords Deep learning · System software · Pervasive computing · FPGA · Adaptive systems

1 Introduction

Since its inception 30 years ago (Mattern 2001; Weiser 1991), ubiquitous and pervasive computing has changed fundamentally how we perceive computers and interact with

them. The ability of a pervasive system to detect the situation of its users in the physical world (Clemente et al. 2020; Sadhu et al. 2019) and to adapt itself to it (Braud et al. 2020; Breitbach et al. 2019) are central building blocks of such systems. Besides context management (Elmalaki et al. 2015) and runtime adaptation frameworks (Becker et al. 2003, 2004; Cervantes et al. 2017; Handte et al. 2012; Herdin et al. 2017), artificial intelligence (AI) and machine learning (ML) algorithms have proven to be of major importance to realize this vision. With them, pervasive systems can e.g. detect user behaviour (Clemente et al. 2020; Radhakrishnan et al. 2020; Zhou et al. 2015), analyse physical environments (Fukushima et al. 2018; Zhang et al. 2017) and optimize system operation (Burger et al. 2020b).

Despite its importance for pervasive computing, AI is typically provided in an ad-hoc manner without comprehensive support of a suitable system software. Instead, it is handled as part of an application and integrated by developers anew for each app. AI components are usually either implemented for the Cloud and Edge, or specifically on embedded devices as isolated solutions. In neither case these solutions are interconnected and do not take into account the specific challenges that pervasive systems pose on them:

The authors acknowledge the financial support by the Federal Ministry of Education and Research of Germany in the KI-LiveS project (Project number 01IS19068A).

✉ Gregor Schiele
gregor.schiele@uni-due.de
Christopher Cichiwskyj
christopher.cichiwskyj@uni-due.de
Stephan Schmeißer
stephan.schmeisser@uni-due.de
Chao Qian
chao.qian@uni-due.de
Lukas Einhaus
lukas.einhaus@uni-due.de
Christopher Ringhofer
christopher.ringhofer@uni-due.de

¹ Embedded Systems Research Group, University of Duisburg-Essen, Duisburg, Germany

Firstly, pervasive system runtime situations are known to be highly dynamic, heterogeneous and often unpredictable. As such, an AI system must be able to handle widely fluctuating applications, data sources, available execution resources and network conditions at runtime.

Secondly, pervasive systems can have very long lifetimes with devices being in operation for decades. The AI system must be able to handle data drift over time as well as new AI techniques that have been developed after a device has been deployed.

Thirdly, pervasive systems often have very strict and often contradictory performance requirements, e.g. with respect to prediction accuracy, energy consumption, reaction time, and privacy. An AI system must allow to optimize its behaviour for such requirements.

In this paper we propose a system platform for pervasive AI applications, that aims to take AI techniques and combine them with approaches from the pervasive computing community, such as Becker et al. (2003, 2004), as well as making pervasive solutions available to AI developers. This allows the AI system to become adaptive and to optimize itself to runtime changes in pervasive execution contexts. At the same time, it relieves developers from the complexity of pervasive computing and allows them to concentrate on their specific AI. Through runtime adaptation and programming abstractions, developers can start with a standard AI, e.g. placed in the Cloud, using input data from a fixed sensor. Then, iteratively, they can extend this approach, e.g. switching to a specialized AI model on an embedded system and taking into account input data from different, changing sensors.

To realise this vision, we use and extend our earlier work on pervasive systems, runtime adaptation and AI, integrating ideas and concepts from nearly 20 years of research.

We propose two main contributions in this paper:

First, we propose a distributed execution environment for AI in pervasive computing that integrates deeply embedded devices, Edge and Cloud servers. To mitigate the limitations of embedded devices with respect to compute power, our system allows to incorporate exchangeable hardware (HW) accelerators for AI that are implemented on local embedded field programmable gate arrays (FPGAs). We call this solution the *Elastic AI*, a distributed AI runtime environment.

As our second main contribution, we show how to use this execution environment to realise adaptive AI systems for pervasive computing, using a variety of structural and behavioural adaptation techniques such as reselection, relocation, and parametrisation for the AI model execution, as well as clustering and data fusion for AI data.

The remainder of this paper is structured as follows. In Sect. 2 we provide the necessary background explaining our assumptions and the resulting challenges and requirements our system must address. We also discuss related

work. Section 3 describes the overall approach and design rationale of our system, after which we present our Elastic AI runtime environment in Sect. 4. In Sect. 5 we describe how a developer can implement an adaptive AI system for a pervasive application using our system. Then, in Sect. 6, we show how to handle varying data sources and data quality. In Sect. 7 we evaluate our approach before concluding the paper in Sect. 8.

2 Background and related work

Before presenting our approach for adaptive AI we first want to provide all necessary background information for the paper. First, we describe our system model for pervasive computing. Then we analyse the specific challenges that pervasive computing poses for AI systems and derive requirements from them. Finally, we discuss our related work.

2.1 System model

We model a pervasive system as a number of mobile end-users that execute pervasive application software on interconnected compute devices. Devices are very heterogeneous with respect to their resources and attributes. They can range from high performance stationary servers in a Cloud centre that are reachable over the Internet, to mid-level Edge devices that are installed in a local environment, mobile battery-powered devices that are carried by users, and low-end (deeply) embedded devices that are integrated into everyday objects with 8bit microcontroller units (MCUs), a few kilobytes of RAM and no operating system support. Many devices also integrate sensors to measure their physical environment and actuators to influence its state. Sensors are themselves very heterogeneous and can range from very cheap, inaccurate ones to very expensive and accurate sensors that measure a variety of phenomena.

Users, applications and devices are not fixed. Users come and go, move between rooms and buildings and between different physical contexts. Applications are started and stopped as needed by users and need differing resources at different stages of their execution. Devices are powered on and off and may be able to provide different sensor data and actions depending on their location. All devices are networked with each other and we assume that each device can exchange information with any other device, e.g. using a wireless local network or the global Internet. However, connectivity can fluctuate, devices may switch between different networks with different performance, and devices might not be able to communicate temporarily.

With respect to applications, we focus on applications that need to work with AI to perform their function. The AI can be realized with a number of different techniques such

as neural networks, unsupervised data clustering algorithms or reinforcement learning. Often, several of these techniques are used in combination with each other. Preprocessing such as filtering or feature extraction may be used, too. The AI is fed input data from one or more data sources, e.g. different types of sensors of local or remote devices. We call the set of all these interconnected parts an AI model.

2.2 Challenges

As discussed before, pervasive computing poses special challenges for AI. In the following, we analyse these challenges in more detail. After that, we derive our requirements from them.

What is special about pervasive computing when it comes to supporting AI?

Unpredictably changing runtime situations As a developer of a pervasive computing application, you have to cope with highly dynamic systems that may change unpredictably and may lead to a huge number of very heterogeneous runtime situations and contexts. Thus, it is not really possible to hard code how the application should be executed in all possible situations. Instead, the application must adapt itself dynamically to cope for unpredictable changes. One moment the Cloud might be available via cheap and high speed communication. Thus, the AI model can be offloaded to it efficiently. In the next moment, the Cloud may not be reachable at all or only very slowly. Thus, the AI model must be executed locally or the application must work without its AI for some time. This results in large fluctuations with respect to the available compute resources. As a result, the complexity of the AI model that can be supported changes.

Fluctuating data sources Typically, in pervasive systems you have a large and changing number of heterogeneous sensors to work with. One moment you might be in a setting where you only have a cheap, low-quality sensor in your mobile device. The next moment, you might have access to sensors from multiple nearby devices, some of them not owned by you. Shortly after, you might have access to a very good sensor that is deployed nearby in the infrastructure (e.g. for air quality). As a result, both the number of potential data sources as well as your data quality may fluctuate quickly and strongly and the AI model has to take that into account.

Long system lifetime Pervasive computing systems are expected to be in use for many years. Many pervasive devices that are embedded into everyday objects cannot be replaced every year without wasting natural resources and money. Instead, the system and its components must be able to evolve after being deployed. They have to adapt to new algorithms and protocols, new security risks, new legal requirements, and new applications. In addition, their physical environment will evolve, too, leading to data drift. This

can degrade an AI model over time and must be taken care of by updating and retraining it after its initial deployment.

Performance Pervasive applications typically have to fulfil very strict performance requirements that need to be balanced. The deployment of a full precision AI model in the Cloud will optimize the resulting prediction quality. At the same time, privacy and safety are major concerns in pervasive computing. We cannot stream all data about a user 24/7 to a server to perform AI computations on it. This would expose the user's daily life completely and is unacceptable. It also invites denial of service attacks—which may lead to physical harm e.g. in medical systems—by degrading or blocking communication between a sensor and its AI server. Therefore, some AI tasks must be executed as close to the data source as possible, in some cases in situ on the sensor device itself. This will also influence the resulting energy consumption and latency.

2.3 Requirements

In this section, we derive the main technical requirements for a system platform for AI in pervasive computing. We derive these from the characteristics of pervasive computing that we discuss before. Similar requirements have been described in previous work for AI systems in pervasive computing (Lalandaet al. 2019) and self-* systems (Bellman et al. 2019).

1. *Optimized deployment across heterogeneous device classes* To respond to the challenges of unpredictable runtime situations as well as performance optimization, we need to have the ability to deploy AI models on all currently available, heterogeneous compute resources. This includes deploying only to the embedded pervasive sensor device itself (e.g. for privacy reasons), to nearby Edge servers (e.g. to reduce latency) or to the remote Cloud (e.g. to work on an integrated view). More complex models must also be deployable in a distributed way with parts of the AI model on each of these device classes.
2. *Continuous evolution and runtime adaptation* Deploying once is not enough. Since runtime situations change in pervasive computing and due to the long system lifetime, the system must support evolving and adapting an AI over time. This includes the ability to update your AI both with new architectures and new training results to ensure an appropriate behaviour of the system for years to come. This is especially relevant when facing data drift or changing application requirements. In addition, the system must be able to redeploy an AI model dynamically and to switch between AI models dynamically to optimize the system.

3. *Data source management* Finally, to cope with fluctuating populations of data sources, an AI system needs to be able to work with different numbers of sensors efficiently, as well as working with fluctuating data quality. This should not require any specific training techniques and should work for any kind of AI model. The AI developer can just use a normal, good quality dataset to train and evaluate the model without taking care of pervasive peculiarities like redundant sensors etc.

2.4 Related work

In the following section we analyse previous and related work for runtime support of AI in pervasive computing.

Today, AI is an important part of many pervasive computing approaches, either executed locally (Fellicious 2018; Fukushima et al. 2018; Krupitzer et al. 2018; Radhakrishnan et al. 2020; Turky et al. 2020; Zhang et al. 2017; Zhou et al. 2015) or in the Cloud (Clemente et al. 2020; Muhammad et al. 2019; Tantawi and Steinder 2019). However, these approaches focus on how to *use* AI in their respective application areas. They do not provide a general adaptation or execution platform for pervasive AI.

In contrast, runtime support for adaptive *applications* in general (as opposed to adaptive *AI*) has been an important topic for the research community in pervasive systems, e.g. (Aberer et al. 2006; Aygalinc et al. 2016; Becker et al. 2019; Caporuscio et al. 2010; Escoffier et al. 2014; Handte et al. 2012). Most of these approaches concentrate on either networked embedded devices (Becker et al. 2004, 2003; Eisenhauer et al. 2010; Kostelník et al. 2011) or the Cloud (Brinkschulte et al. 2019; Guinard et al. 2010; Mahn et al. 2018; Naber et al. 2019), thus not fulfilling our first requirement, to provide a runtime environment for all device classes.

This has changed only recently. Barnes et al. (2019) and Lalanda et al. (2017, 2018) provide execution platforms for adaptive pervasive systems that take into account all device classes. However, similar to earlier approaches, they do not provide any specific system support for adaptive *AI*. Therefore the development effort required for AI components remains high.

As a third group of related work, there are a number of systems, most prominently in the domain of the Internet of Things, that are specifically focusing on runtime support for AI components in networked systems, e.g. Amazon (2021), Kim and Kim (2020), Li et al. (2019) and Microsoft (2021). However, these focus purely on running AI models in the Cloud or Edge and do not support the level of dynamism necessary for pervasive application cases.

Designing AI models for deeply embedded devices is another important research area. This requires model optimisations to cope with the resource constraints imposed by these devices. Different approaches focus on different types

Fig. 1 Approach overview

of embedded hardware. To execute models on embedded MCUs, highly optimised software implementations of AI models are required. Approaches such as MCUNet (Lin et al. 2020) or MobileNetV1/V2 (Howard et al. 2017; Sandler et al. 2018) address this by optimising Deep Neural Networks (DNN). As an energy-efficient alternative, embedded FPGAs can be used as hardware accelerators for DNNs (Wang et al. 2019; Musha et al. 2018; Venieris and Bouganis 2016, 2017, 2019; Yang et al. 2019; Zhang et al. 2015). Due to an embedded FPGA's limited available resources to instantiate circuit designs, further optimisations are required. Examples are binarisation or quantisation (Han et al. 2015; Iandola et al. 2016; McDanel et al. 2017), pruning unnecessary neuron inputs (Han et al. 2015; Roth et al. 2020; Hassibi and Stork 1993; Yang et al. 2017) or reducing the mathematical complexity of the underlying models (Rastegari et al. 2016; Wang et al. 2019). Other approaches aim to develop special ASICs for neural processing units (NPU), e.g. Google's tensor processor unit (Jouppi et al. 2018), or CONV-SRAM (Biswas and Chandrakasan 2018).

While these approaches are crucial to create embedded AI models, especially on deeply embedded devices, they do not address how the resulting models are deployed, executed, and adapted in a concrete system.

Although no previous work has provided full system support for runtime adaptation of pervasive AI, some approaches have proposed solutions for different aspects of AI adaptation.

Federated learning allows to adapt an AI by retraining it at runtime (Ek et al. 2020, 2021; Saeed et al. 2020; Konečný et al. 2016). Existing approaches do not tackle the aspects of AI software component deployment.

Cox et al. (2021) propose a generic execution framework for DNNs on Edge devices that includes a memory-aware scheduler for multiple, concurrent DNNs.

Thus, they adapt the order in which multiple AI models are executed. The framework is limited to a single device and does not address distributed AI deployments.

Houzé et al. (2020) present a component-based decentralized AI system for smart homes that allows to adapt an AI model when devices join and leave. However, the main focus of their work is on how to use this for explanatory AI. They do not provide details on their runtime environment or how to integrate deeply embedded devices.

A different field of research (Yu et al. 2018; Guerra et al. 2020; Jin et al. 2020) has delved into creating AI models that are able to adapt their internal state according to the input data. While we consider this an additional powerful tool to create appropriate AI models for pervasive applications, the range of challenges a single network can address will be limited in contrast to the vast and potentially drastic context changes that can occur in a pervasive system. As such, while an adaptable AI model (Yu et al. 2018; Guerra et al. 2020; Jin et al. 2020) may reduce the necessity for switching between models, we still consider that a necessity for truly pervasive systems.

To conclude, while a lot of work has been done on runtime adaptation in pervasive computing in general, supporting adaptive AIs in such environments has not been explored enough. No system exists that fulfils all our requirements, either lacking the ability to support Cloud, Edge and embedded devices, (b) runtime support for AI, or (c) support for pervasive environments.

3 Approach overview

In this section we present our proposed approach to enable adaptive AI in pervasive computing. We give a brief design rationale for our approach, discussing major design choices. Then, we describe our open hardware platform that we are using as the embedded device target platform for our work. In the next sections, we discuss each part of our approach in more detail.

3.1 Design rationale

A conceptual overview of our approach is given in Fig. 1. We essentially have to provide three parts:

First, we need a way to *specify adaptive AI models*. In our system, an AI model is a graph of interconnected AI components as depicted in Fig. 1a. The graph contains all parts and dependencies of an AI model, e.g. necessary input data, data filters and pre-processing, feature extraction and classifier. It essentially specifies the granularity of the model and gives

restrictions on how it can be adapted. We decided to realise AI graphs programmatically, e.g. by allowing the developer to program AI components in isolation and to connect them with each other in code at runtime by binding to remappable URIs. This provides loose coupling between them. We do not support a specific declaration language for AI graphs. This is similar to the approach taken by established AI frameworks like TensorFlow and thus well-known to AI developers.

Second, we need an *execution environment* to run adaptive AI models (see Fig. 1b). To fulfil our Requirement 1, namely, to distribute AI models across all device classes, the execution environment software allows developers to deploy and run their own AI models as well as pre-existing ones on a large span of devices. Devices range from embedded devices, to Edge servers and the Cloud. To better address the specific characteristics of these very different target platforms, we subdivide our execution environment into two distinct, cooperating runtimes: one for Edge and Cloud devices, and another one for deeply embedded devices. Following from our discussion in Sect. 2.4, deeply embedded devices typically lack the resources to execute meaningful AI models locally without draining their battery. Therefore, we decided to enhance such devices with low-cost embedded FPGAs. These can instantiate AI algorithms efficiently in hardware and act as local accelerators for high speed AI. Since they can be updated with new accelerator code after deployment, this also allows us to support continuous evolution of the embedded AI. Because we did not find a suitable open hardware platform for such devices, we developed our own, which we briefly present in Sect. 3.3. More details about our distributed execution environment can be found in Sect. 4.

Third, we need to support *adaptation mechanisms and strategies* that developers can use (see Fig. 1c). This enables us to fulfil our Requirements 2 and 3. There are many different adaptations that a system can perform [see e.g. Krupitzer et al. (2015)]. We decided to support (1) adaptations on the AI model and (2) adaptations on the input data and data sources. For the AI model, we focus on updating, reselection and relocation of AI models, namely the nodes in the AI graph. Updating allows to upload new code to devices to replace outdated components. Reselection is used to exchange a component in the graph, e.g. to use another implementation for filtering or classification. We can also reselect data sources, e.g. by receiving data from a different embedded device. Relocation modifies where a given component is executed, e.g. moving it from a Cloud to an Edge device. We provide more detail about this in Sect. 5.

Executing an AI model is useless without the right input data. Therefore, we provide mechanisms and algorithms for data adaptation and data source management (see (f) in Fig. 1). We offer data clustering algorithms to map required input data to measurements of groups of sensors that are

combined with a suitable sensor fusion algorithm. This allows to use measurements from changing numbers of data sources as input into an AI model without the need to train different models. More detail about this can be found in Sect. 6.

3.2 Automatic vs manual adaptation

Note, that in contrast to many other systems, our approach does not focus on providing fully automatic adaptation support. Instead, we offer partially automated adaptation and rely on the developer for the rest. This is due to two main design considerations.

First, we want to support very resource-restricted embedded devices. These are not powerful enough to execute complex adaptation strategies on their own, e.g. if the Edge or Cloud is not reachable. Therefore, in our embedded runtime we rely on the developer to use our adaptation mechanisms in their code for a specific—application-dependent—adaptation strategy. For Cloud and Edge devices, which have much more resources, we provide automatic adaptation for placing components and managing their lifecycle.

Second, in our research we experienced that fully automatic adaptation strategies are often not the best solution for real systems. Developers have specific goals in mind when designing their applications. They know when to best adapt, when not to, and how to notify the user about a planned or ongoing adaptation, either to reduce annoyance or for liability reasons. Sometimes an adaptation must take into account the potential for partial failure, especially in embedded systems that may be safety critical. This is very difficult to automate correctly in all cases if we have to provide safety guarantees.

With these considerations in mind, we decided to provide automatic adaptation support for cases where we can do so without impacting the available resources too much and which are not visible to the user. This is the case mainly for placing and relocating AI components on computers in the Edge and Cloud as well as for adapting input data. Automating these tasks already reduces the workload on developers. For additional adaptation tasks, e.g. updating code or reselecting an AI model or data source, we restrict ourselves to providing the tools to enable developers to program their own adaptation strategy as easily as possible.

3.3 Open hardware for FPGA-based embedded AI

As discussed before, part of our requirements is to execute AI models locally on a pervasive sensing device if necessary, e.g. due to privacy concerns or because communication is too slow to send raw data to a Cloud-based AI.

Fig. 2 The Elastic Node v4

Because conventional CPU architectures cannot execute AI models with good performance, alternative architectures are developed, such as custom neural processors units (NPU) (Jouppi et al. 2018) or advanced RAM architectures (Biswas and Chandrakasan 2018). NPU-based approaches however lack the necessary flexibility to adapt to large changes in the environment.

Embedded FPGAs have become a flexible solution for this by implementing AI models as accelerators (Venieris and Bouganis 2016, 2017, 2019; Yang et al. 2019; Zhang et al. 2015). We believe that with the current generation of embedded FPGAs they are both powerful as well as cheap and energy-efficient enough to be integrated into many pervasive devices, including mobile sensors and actuators. This also enables the local AI to evolve over time. AI models implemented as an FPGA hardware accelerator can be instantiated as required, the FPGA can reselect between different available models directly at runtime and models can easily be updated to newer or different implementations to adapt to new and unexpected changes in the application for years to come.

Therefore, we decided to provide full support for such embedded FPGAs as part of our AI runtime.

Using only an FPGA in a pervasive device, however, is not a feasible solution, due to the FPGA's comparatively high power consumption when performing basic tasks. Many tasks in a pervasive device do not require much computational power, e.g. reading out a sensor value and sending it to a processing service. These can therefore be handled by a classic device design containing something as small as an 8bit MCU. As a result, we argue that a future, AI-enabled pervasive device must include heterogeneous compute cores, including at least a low-power MCU and an embedded reconfigurable FPGA, which must all be interconnected fast and with low overhead to enable them to collaborate efficiently.

Since we were not aware of an open hardware board that fulfils these requirements, we developed our own board, the *Elastic Node* (Burger et al. 2017; Schiele et al. 2019). An assembled Elastic Node in its fourth version can be seen in Fig. 2. It combines a classical 8bit MCU with an energy efficient embedded (Xilinx Spartan 7) FPGA. An application is split across the MCU and the FPGA. For most regular tasks, the MCU handles them while the FPGA is powered off, conserving as much energy as possible. However, if a more computationally intense AI task is scheduled, the FPGA can be powered on and a tailor-made AI accelerator circuit is instantiated. This accelerator can then process the AI task at a much faster rate and more energy efficiently than doing so on the MCU. To use this hardware platform, we included support for it into our distributed AI runtime, which we describe in more detail in the next section.

4 Elastic AI: a distributed AI runtime environment

In this section we discuss our distributed runtime environment for AI models. Most importantly, this runtime environment supports embedded as well as Edge and Cloud devices to fulfil our requirement 1 (making full use of all heterogeneous resources). Thus, it enables to deploy and execute AI models that are distributed over all these device classes. To do so, we developed two distinct but integrated runtime systems: one for Edge and Cloud devices, another one for (deeply) embedded devices.

The main reason for separating these sub systems is that they have to cope with very different requirements. Software on embedded devices has to work efficiently with very restricted resources. We target tiny 8bit micro controller units (MCUs) with some kilobytes of RAM that are battery operated. Such low power MCUs are usually programmed in low level languages like C without full-fledged operating system support and have to guarantee realtime and safety properties. Therefore, dynamically loading and executing code is often out of the question. Cloud and Edge devices on the other hand have comparatively massive amounts of computation, storage, and energy resources. Realtime is typically a lesser concern and robustness can be achieved using high levels of redundancy between devices. Programming is often done with high level languages such as Java or Python. Virtual machines, containers and dynamic orchestration are standard features. In the following we first describe the runtime for Edge and Cloud servers. Then, we present the runtime for deeply embedded systems.

Fig. 3 Edge and cloud runtime

4.1 Edge and cloud runtime

As depicted in Fig. 3, our Edge and Cloud runtime is composed of two kinds of software components that can be deployed dynamically. First, a set of *digital twins* that can be used to implement application logic and to extend the system, and second, a collection of *system services* that implement system-wide management functionality. Digital twins can e.g. represent physical entities such as a smart pervasive device, a room, or a user. System services provide e.g. support for intra-system communication and control, as well as fluctuating device populations.

All of these system parts are implemented as containerized microservices that can interact with each other using a resource-oriented communication abstraction. They are orchestrated automatically by Kubernetes. We specifically decided against developing our own orchestration framework. Existing technologies are mature, provide good performance and are well supported by developers and tools. Using Kubernetes, we only have to provide a YAML descriptor detailing our deployment and Kubernetes takes care of all placements. It offers life-cycle management for container-based system components, monitors their states and if any of them shuts down unexpectedly, it restarts or relocates the container automatically. New twins can be added at runtime by deploying them with their own descriptor, either by providing the descriptor directly to Kubernetes or by having Kubernetes download it together with its container.

Kubernetes however has two shortcomings that we must address. First, it is not viable for deeply embedded systems. Therefore, we restrict its usage to larger devices, which may include larger embedded systems, too. We provide system services that allow to connect to our embedded runtime, e.g. to exchange data, and to manage devices in the embedded runtime, adapt them and deploy code on them. As a second shortcoming, Kubernetes does not manage fluctuating device populations directly. Therefore, we add system services to detect and integrate such devices dynamically.

4.1.1 System services

We provide a set of fixed system services consisting of: (1) the message broker, (2) the URI resolver, (3) the bootstrapper, (4) the translation services, and (5) the embedded component controller. In the following we will discuss briefly each service individually.

(1) *Message broker* All communication is based on the concept of interacting remote resources that are identified using URIs. Accessing a URI is mapped to sending a message to a twin. We will refer to this action as calling a URI. To guarantee message delivery we rely on a centralized message broker with a topic based publish/subscribe paradigm. In its current form we are using the MQTT protocol as our messaging system. Its pub/sub architecture allows for ideal communication flow and offers a flexible quality-of-service architecture. The centralized message broker can easily be operated in a high-availability configuration with multiple instances to avoid single point-of-failure and load balancing.

(2) *URI resolver* With everything being accessed via a URI we want to offer the ability for developers to compose multiple components into a higher concept. This can prove difficult when trying to keep track of all involved parties, especially in a highly volatile pervasive computing system in which devices enter and leave frequently. The URI resolver maps a URI call to the correct receiver. When a URI is called, the resolver is queried whom to contact and provides all necessary information to execute said call.

(3) *Bootstrapping service* The bootstrapper has two main functions: a presence service and a directory service. The presence service is the first point of entry for every device and its digital twin. They register with the service and provide a self-description of themselves, which is stored in a repository. The presence service establishes a link between a device and its twin, starting and stopping a device twin if needed. It also notifies the URI resolver about all new URIs and their mappings for a new device/twin pair. Finally, the presence service provides a heartbeat mechanism to check for unexpected system leaves to keep track of all registered devices/digital twins. The directory service uses the presence service to allow clients to search for digital twins that provide specified functionalities using semantic reasoning.

(4) *Translation services* Communication between different components in the Edge and Cloud runtime is based on MQTT, which requires IP. Deeply embedded devices however are often unable to implement a full IP stack. For such devices we provide Translation Services. Translation Services can be implemented by a gateway, a dedicated piece of hardware that supports multiple communication protocols and acts as a proxy between them. Relaying messages from and to the Edge and Cloud runtime via such a proxy happens fully transparently to both device and digital twin.

Ideally, the gateway is deployed in the Edge, acting as a bridge between the Cloud and the embedded hardware.

(5) *Embedded component controller* The embedded component controller provides over-the-air (OTA) update functionality to physical devices as well as their corresponding twins. Embedded software can be updated, and in case of our Elastic Node new FPGA functionality in the form of bit files can be supplied to the FPGA to extend or exchange the available set of accelerators on the device. It manages the retrieval of updates, e.g. by a provided download link, and transparently updates devices.

4.1.2 Digital twins

As mentioned before, the main interface for developers to access our system is through the use of so called digital twins. A digital twin offers a resource-based API to access other system parts. It can represent physical or virtual entities in the system. In addition to this basic twin, we provide specialized twin types that provide more powerful APIs to developers. We offer a pre-defined set of classes and annotations you can use out of the box to reduce the amount of boilerplate code.

A *device twin* represents a hardware device (and its sensors) that is embedded into an environment. Our system supports linking a device twin to its device and provides system services for synchronizing their life-cycles. A device twin is started automatically by the *Bootstrapping service* when its device joins the system and is stopped when the device leaves. To hide fluctuating connectivity, a device twin can predict sensor values for a currently disconnected device. A device twin can also be asked about historical data or predictions of future states.

A *composite twin* is a twin that combines other twins. This can, e.g. represent a room that contains several devices. This way, a client does not need to know which devices are currently in a room to get sensor data from them. Instead, it can contact the room's composite twin and ask for sensor data. The room's composite twin can use the *Bootstrapping service* to keep track of which devices are currently located in the room and can contact their device twins to get sensor readings. Of course, composite twins can also use other composite twins, forming a multi level hierarchy.

A *data twin* is a twin modelling a stream of sensor data as input data. It provides a configuration API that allows other twins to specify the required data quality. If this quality cannot be provided, it notifies its clients. Internally, a data twin can attach to a specific device twin and request data from it or it can use our data adaptation algorithm to use fused sensor data from multiple sources, e.g. a composite twin combining multiple device twins for data gathering.

An *AI twin* abstracts an AI model. It can be implemented as a basic twin, including the whole AI model as a single

Fig. 4 The embedded runtime (excerpt) for MCU-FPGA devices

monolithic component. To do so, our system provides the ability to access existing AI frameworks like TensorFlow and to execute an AI model in them. As an alternative, an AI twin can be a composite twin. In this case, the AI is modelled as a set of different twins that are wired to each other. Thus, e.g. pre-processing can be modelled as one twin, feature extraction can be modelled as a second, and classification can be modelled as a third one. The composite AI twin can then link these twins and reselect between different twins at runtime as it chooses, e.g. to cope for new devices joining, changing network connectivity or fluctuations in data quality. It can use the *Bootstrapping service* to get a current view on AI twins to choose from.

Note, that to reduce development effort, we also provide a set readily-implemented twins, e.g. data twins that include data clustering or sensor fusion. A developer can use these as part of her application. We describe these twins in more detail in Sects. 6.2 and 6.3. You can find a more thorough description of the runtime in our previous work (Burger et al. 2020a).

4.2 Embedded runtime

Developing software components for deeply embedded devices can be a challenging task, due to the lack of resources. This results in having to develop an application in a bare-metal approach, e.g. without an underlying operating system. For hybrid hardware such as the Elastic Node, the complexity increases even more, as a developer has to handle the hardware interactions directly between both processing units as well as organising how to pass data between the application components distributed on MCU and FPGA.

To simplify this process as well as to provide tools to integrate deeply embedded devices into the Elastic AI, we created a software suite written in C using a bare-metal design approach. We call this the Elastic Node Middleware (Embedded Systems Department UDE 2019). This system software consists of components, that are deployed on both the MCU and circuit FPGA and are aimed to be as resource

efficient as possible to accommodate the limited available resources. With this, developers are able to create their own application logic on a deeply embedded device and can incorporate newly developed FPGA-based AI models for their specific application, all while reducing the total development overhead. So far, designing and synthesising FPGA-based AI models is supported by existing tools and toolchains, e.g. by using Xilinx Vivado (Xilinx 2021). Further design support is subject to our ongoing research.

While our middleware provides a wide range of services, e.g. to perform measurement experiments to estimate energy consumption for your application, we will focus on the services required to support AI components on the device. An overview over these can be seen in Fig. 4. A complete overview can be found in Burger et al. (2020a).

Due to the harsh resource limitations, especially on the MCU side, the Elastic Node Middleware focuses less on creating a runtime that allows to dynamically create the appropriate behaviour. Instead we strive more towards using code generation, where we create a fitting, static embedded application component. These components can then be deployed as a whole using over-the-air updating techniques.

Note that embedded devices that are powerful enough to run operating systems such as Linux are also able to execute Kubernetes. Therefore, such embedded devices are, from a development approach view, part of our Edge and Cloud runtime and do not require an extra bare-metal development approach.

4.2.1 MCU services

On the MCU we offer the following services:

Remote resource framework (R2F) To provide an abstracted view towards the communication protocols and to provide a resource-oriented interaction scheme with other remote components in our system we offer the remote resource framework (R2F) service. R2F offers a self-description of the current system capabilities. Through the R2F, remote services can access data, control different device actions such as the power state, and deploy components on it. R2F allows developers to both write device-centric applications, that can react to external requests as well as cloud-centric applications that have full control over the devices.

Hardware component manager The hardware component manager service allows developers select, which FPGA accelerators to instantiate out of a set of currently available accelerators. It reduces the complexity of the MCU-FPGA interactions for loading and instantiating the right accelerator and handles the communication between MCU and FPGA.

Offloading manager As soon as another, remote component in our system requires a task to be executed locally, e.g. by running a specific accelerator, we can handle these

requests with the Offloading Manager. It detects which FPGA accelerator is required to fulfil a request and triggers the reconfigurations through the hardware component manager.

OTA updater To support applications with lifetimes common in pervasive computing, both the soft- and hardware components will have to be updated to continue offering functionality adequate to the current system goals and the surrounding environments. The Over the Air (OTA) Updater service offers resources, that enable developers to swap software components on the MCU as well as FPGA accelerators, that are then incorporated into the device application at runtime.

4.2.2 FPGA services

On the FPGA we offer the following services:

Communication manager To provide a component equivalent to the MCU's hardware component manager, we use the Communications Manager. It handles the incoming and outgoing data exchanges between MCU and FPGA. The received data can either be passed directly to the core accelerator logic, or to a skeleton component that translates between the unified interface and the individual structure of the accelerator logic.

Reconfiguration control While we assume that switching between accelerators is aimed to be mainly managed by the hardware component manager, a reconfiguration of the FPGA can be started from within the FPGA as well. This can be used by accelerator designs that are subdivided into multiple smaller designs which are executed sequentially. This allows designs to be extended to support more complex accelerators.

4.2.3 Stubs and skeletons

While developers can use the MCU and FPGA services directly, they still need to provide information which is specific to each accelerator: where the circuit is stored in the flash chip of the Elastic Node hardware and how the data should be exchanged between the MCU and the FPGA side. To make it easier for developers to incorporate their own user defined application components and AI models on our deeply embedded device, we further simplify this process by using a stub/skeleton approach. Stub and skeleton lie between the software component or hardware accelerator and the middleware on either side, as shown in Fig. 4. It abstracts from and isolates the specific deployment details for a given accelerator, providing a single, semantically unambiguous function towards the application itself.

Listing 1 Example stub implementation to access a CNN accelerator on the FPGA

```

1  #define LOC_CNN 0x0
2  #define INPUT_ADDR 0x100
3  #define RESULT_ADDR 0x101
4  uint16_t cnn_execute(uint8_t input){
5      elasticnode_fpgaPowerOn();
6      elasticnode_reconfigure(LOC_CNN);
7      elasticnode_writeData(INPUT_ADDR,
8          input, sizeof(input));
9      elasticnode_execute();
10     while(!elasticnode_isDone()){
11         uint8_t result;
12         elasticnode_read(RESULT_ADDR,
13             &result, sizeof(result));
14         elasticnode_fpgaPowerOff();
15         return result;
16     }

```

An example for a stub implementation can be seen in Listing 1. Within the stub we define the location of the corresponding bitfile in flash, as well as the explicit addresses to our memory mapped communication interface to the FPGA. These are meant to mirror the expected communication behaviour on the accelerator side and should be designed together with them. For the developer, calling a CNN which is locally available on the FPGA is then reduced to a simple, single function call.

To similarly simplify the integration of an accelerator with the FPGA middleware services we propose using a skeleton. It bridges the specific behaviour of the accelerator circuit with the unified middleware communication interface. An example entity for the CNN skeleton, counterpart to the above stub implementation can be seen in Listing 2.

Listing 2 Example Skeleton interface for a CNN accelerator in VHDL

```

1  entity CnnSkeleton is
2  Port (
3      -- control interface
4      clock : in std_logic;
5      reset : in std_logic;
6      busy  : out std_logic;
7
8      -- memory mapped interface
9      -- request a variable
10     rd : in std_logic;
11     -- request changing a variable
12     wr : in std_logic;
13     data_in : in uint8_t;
14     address_in : in uint16_t;
15     data_out : out uint8_t;
16 );
17 end EchoSkeleton;

```

Due to the memory-mapped interface on the MCU side our skeleton can receive a read or write command to a specific address to request an action based on the target address.

Table 1 FPGA resource overhead on a Xilinx Spartan 7 XC7S15

Resource	Used	Available	Percentage
Slice registers	88	16,000	0.55
Slice LUTs	63	8000	0.79
DSP	0	20	0.00
BlockRAM	0	10	0.00

Fig. 5 Reduced embedded runtime for conventional devices

The stub/skeleton structure can be derived fairly easily from the interface description of the hardware accelerator on the FPGA and from the deployment information where its corresponding bit file has been stored. Thus, we aim to auto-generate the stubs and skeletons in the future.

4.2.4 Overhead

To put our system components into context we want to briefly give an overview over the resource consumption of this embedded runtime on both the MCU and FPGA side. On the MCU side the software components are made up of 2350 lines of code and require 2.1 kilobytes of RAM and 16.1 kilobytes of program flash. For the FPGA libraries, the resource consumption can be seen in Table 1 in relation to the available amount of resources on a low-power FPGA, the Xilinx Spartan 7 XC7S15, that is currently also supported on the Elastic Node platform. Our libraries only require 0.79% of the available registers and 0.55% of the Look-up Tables.

4.2.5 Integrating conventional deeply embedded devices

While this software suite was designed with the Elastic Node Hardware in mind, its different libraries are not implemented to run only on the Elastic Node Hardware but a range of different microcontrollers. This is achieved thanks to appropriate hardware abstractions, making these libraries platform agnostic. It enables developers to use parts of the Middleware even on conventional deeply embedded devices, allowing them to integrate them into the Elastic AI in just

Fig. 6 Overview of the components of the basic AI model application including where each of the different parts is run

the same way. An example of such a structure can be seen in Fig. 5. The behaviour of the different libraries remain unchanged.

5 Adaptive AI models

After presenting our distributed runtime environment in the last section, we now describe how to use it to implement and execute an adaptive AI. As discussed before, an AI model in our system is essentially a graph of interconnected AI components.

Developers have to create (or reuse) components for the different runtimes and assign URIs to them. More concretely, an adaptive AI model can consist of a combination of (1) data twins and AI twins in the Cloud/Edge, (2) R2F callbacks and embedded software on the MCU, and (3) AI accelerators on the FPGA. If a new device is used, then the developer may also need to provide a device twin for it.

Components in a graph communicate by sending messages to each other's URIs. Each graph is managed by one AI twin, which can search and select all other parts, link them together by injecting URIs to them or map URIs to different twins.

In the following we describe how you can use this to specify increasingly complicated and flexible AI models, starting with a basic model without adaptivity, showing how to change its design during continuous development and finally extending it to include runtime adaptation for the deployment and selection of model parts. This shows how we support our Requirement 2 for continuous evolution and runtime adaptation.

Fig. 7 Communication flow for the basic AI twin. The AI model requesting an audio stream starts a cascade of (transparent) interactions leading to the device twin requesting data from the device by

means of Translation Service and buffering it to send to the data twin. Note: The Translation Service as a separate entity is omitted for readability. If used it is referenced by its acronym *PTS*

5.1 A basic AI model

For a basic AI model (we call it `EIP://twins/ai/basic`), you first need to specify the input data for the AI. You do this with a data twin, e.g. `EIP://twins/data/basic`. The AI model, as well as the data twin are hosted and orchestrated in the Cloud. Figure 6 depicts an overview of all the twins as well as their deployment location used in this example. Despite being in the Cloud, the data twin provides access to a stream of input sensor data from one or more sensors on one or more devices. In our basic case, to access e.g. audio data from a microphone on an embedded device X, you configure the data twin to request data from X's device twin by handing it a corresponding URI, e.g. `EIP://X/audio`. In Fig. 6 requests are marked as arrows with solid lines. The URI to request audio data will, internally, be mapped to the actual device twin's URI `EIP://twins/X/audio`. This subtle abstraction allows you to interchange your data sources in the long run without needing to change access patterns. The communication flow of the different components is shown in Fig. 7. It entails all URI abstractions and demonstrates the communication behaviour in our system. The device twin requests and buffers the audio data from the embedded device using the URI `EIP://devices/X/audio`, which represents the data on the device itself.

Note that the data twin can also access the data on the device directly. However, we rely on the device twin to ensure data buffering, such that data is requested from the device only once, thus reducing its energy consumption. In this most basic form, a data twin is just a facade, providing the AI system with an API to access data and otherwise just forwarding data from the device twin. In more realistic, more complex scenarios, the data twin is more powerful. We discuss this in more detail in Sect. 6.

Listing 3 shows an excerpt from the device twin of X. To make it a device twin you simply have to extend our base *AbstractTwin* class. It offers already implemented functions for initialising and starting the twin, and only needs basic parameters, like the URI of the twin itself. By starting the twin, it subscribes to its own URI automatically. The developer then can specify handler functions that are called when a specific URI is accessed. To simplify this, we offer an annotation-based scheme. As an example, if the URI `twins/X/audio` is accessed on the device twin, then the annotation `@EIPMessageHandler("/audio")` marks the function `audio` to be called. The function initially checks if the requested data is already buffered locally. Otherwise, it first requests the data from its physical device by using the device URI. Then, it forwards the buffered data to the data twin. The

`callURI()` function is also provided by our API and allows calling remote URIs through our platform.

Listing 3 Device twin for device X.

```

1 DeviceTwinX extends AbstractTwin{
2   init("twins/X");
3   start();
4
5   @EIPMessageHandler("/audio")
6   public void audio(EIPMessage message) {
7     if(buffer.isEmpty())
8       buffer = getDataFrom("EIP://devices/X/
          audio");
9     callURI(message.senderURI, buffer);
10  }
11 }
```

Since the embedded device cannot communicate directly via IP, requests sent to it are routed over a Translation Service, which reformats the message and sends it to the device using the best available transmission technology. The Translation Service is ideally located in the Edge (cf. Fig. 6). It is identified by the URI Resolver, which rewrites any URI identifying the embedded device to point to the Translation Service instead (e.g. `PTS://X/audio`). Note that the Translation Service sends the original URI to the device as part of the message payload.

On the device, R2F receives the message and looks up the callback function that should handle it (based on the included URI). For raw data access, this function is usually already provided by the device developer, similar to Listing 4.

Listing 4 R2F Handler providing access to raw audio data

```

1 void handler_rawAudio(uint16_t requestId)
2 {
3   uint8_t data[bufferSize];
4   microphone_read(data, bufferSize);
5   r2f_response(requestId,
6     data, bufferSize);
7 }
```

The function implementation is simple and can even be auto-generated easily. It is called when the message for its corresponding URI is received by R2F on the embedded device, reads a chunk of audio data from the microphone, packs it into a response message and returns that message to the Translation Service in the Edge, where it is reformatted and forwarded to the device twin in the Cloud. The device twin buffers the data and sends it to the data twin. With this, the input data is available at the data twin and can be used.

The corresponding main function of the software component on the embedded device is shown in Listing 5. Again, a code generator could auto-generate this code quite easily.

Listing 5 Initialisation and duty cycle handling

```

1 int main(void) {
2   r2f_registerHandler(
3     handler_rawAudio,
4     "EIP://devices/X/audio");
5   elasticnode_init();
6   microphone_init();
7   platform_init_sleep();
8
9   while(1){
10    r2f_dutyCycle();
11    elasticnode_dutyCycle();
12    platform_sleep(SLEEP_LENGTH);
13  }
14 }
```

All handlers have to be registered initially at boot time together with their corresponding URI. Due to the resource constraints and the therein resulting lack of true multithreading capabilities, all software components have to be incorporated into the main loop of the MCU's single software component.

As your next step as a developer, you need to specify the AI model itself. You do this with an AI twin (e.g. with URI `EIP://twins/ai/basic`). Since our basic model does not allow adaptation and is executed fully in the Cloud, the AI twin can include the whole AI model as a single monolithic component. Our system provides an API to access existing AI frameworks like TensorFlow and to execute an AI model in them. TensorFlow is included into the container of the AI twin and thus, is placed and run automatically by Kubernetes.

To get its input data, your basic AI twin uses the URI `EIP://twins/audio` which will be translated to the previously specified data twin `EIP://twins/data/basic` and data messages are automatically routed to it.

On the embedded device you only need to specify a single callback function and a simple main function. Note that this is only necessary if you want to use a new device that has not been integrated into the system, yet. On the Edge and Cloud side, you need to program a few lines of code for the AI twin, the data twin and maybe a device twin (again, if you use a new device). Almost all functionality is provided by APIs and system services that are already available. Deployment and runtime management is done automatically by Kubernetes, except for your new embedded code which you can deploy on the embedded device remotely with our OTA Updater.

5.2 Continuous development: using an AI model on an embedded device

With the first version of the AI solution deployed, a more advanced version might be already in development. At some point, you may decide that it is actually more efficient to execute the AI model on the FPGA on the embedded device itself, instead of sending raw data to the Cloud. This can reduce communication overhead but also can increase data privacy. To do so, you need to modify the AI twin, add a new data twin (or modify the existing one), program a new R2F callback function, and develop a new AI accelerator for your AI model on the FPGA. All except the last step are rather trivial. The AI twin must be modified to simply forward the results of a data twin instead of executing the AI model directly via TensorFlow. The data twin now represents the results of your embedded AI model instead of the raw sensor data. This new data twin is reachable via a new URI. The new data twin is similar to your earlier one, using the platform's system services to connect to the device's twin and receive data from it.

For the AI accelerator on the FPGA, we are establishing an online library that allows to share AI models for FPGAs with others. If the needed AI model is available in this library, then the developer can simply download it (more specifically an accelerator for it) and use the OTA Updater to deploy it in the flash memory of the embedded device. Otherwise, the developer needs to program a new accelerator in VHDL and make it available to our system. We provide a number of helper tools and code to ease this process but nevertheless, a VHDL expert is needed for this step. More details on this can be found in Schiele et al. (2019). As part of our ongoing work, we are actively working on a development toolchain that will allow to deploy TensorFlow models directly onto embedded FPGAs by generating the required VHDL code automatically.

As a final step for moving to an embedded AI model, you need to extend the R2F mapping on the embedded device by adding a new callback function for a new URI representing the AI result. An example for this can be seen in Listing 6. Initially, the function is identical to our earlier, raw data example. You first request audio data from the microphone. Then, you need to use your embedded AI accelerator on the FPGA, e.g. a CNN. This is where our embedded middleware comes into play. Each AI accelerator in our system provides a C stub that—using the embedded middleware services—automatically activates the FPGA, reconfigures it to the specified AI, sends input data and receives results. Then it deactivates the FPGA to save energy. All this is hidden behind a single C function call of the stub (see Listing 1). The result is then packed into a response message and send back to the remote caller.

Listing 6 R2F Handler for fully embedded AI

```

1 void handler_embeddedAI(uint16_t id){
2     uint8_t data[bufferSize];
3     microphone_read(data, bufferSize);
4     uint32_t result = cnn_execute(data,
5     bufferSize);
6     r2f_response(id, result, sizeof(
7         uint32_t));
8 }
```

Moving from the old version with a Cloud AI and the new one with an embedded device now only requires to deploy the new embedded code with our OTA Updater. Since the old URI for raw data is still available, you can in parallel continue using the Cloud AI. Figure 8 shows the communication flow in which both data twins are available and only the URI mappings are changed at runtime. In our case the user triggers the OTA update to push her new code to the embedded device. Kubernetes starts the additional data twin without interfering with system operation. This can happen automatically or at the users request. To actually reselect, you could stop the AI twin, replace it with the new one and tell Kubernetes to start it again. Alternatively, you can run both AI twins in parallel and simply specify to the URI Resolver to map the URI of the old AI twin to the new one, as depicted in Fig. 8. This immediately routes all new requests to the embedded solution. Similarly, you can continue developing your AI system and keep deploying new versions, always providing the latest technology to users.

5.3 Runtime adaptation: relocation and reselection

So far we have seen how to use our platform to specify monolithic AI models for the Cloud as well as for an embedded FPGA. We also showed how to reselect between them during development. At runtime, we can provide further flexibility by allowing the system to dynamically reselect different configurations as needed. As an example, you can decide to have three runtime versions and to let the system reselect between them e.g. depending on the available communication interface or relocate depending on a device's or server's status. While your user is at home in her own WLAN, you want to use the Cloud-based AI. When she leaves her home and is on the move, her embedded device connects to the mobile network and switches to the embedded AI model, greatly reducing data transmission cost. As a third scenario, if she carries her mobile phone with her, the embedded device connects to it via

Fig. 8 Communication flow for the extended AI twin. In this scenario Kubernetes starts the new Data Twin (CNN Data Twin) and the user provides a CNN accelerator to the embedded device. At runtime the AI model remaps the URI for audio data to receive the CNN data

from the device, rather than the raw audio data. Note: The Translation Service as a separate entity is omitted for readability. If used it is referenced by its acronym *PTS*

BLE. The phone is powerful enough to execute our Edge and Cloud runtime and can thus become a member of our system. Components can be relocated to the phone. The FPGA on the embedded device is used for preprocessing and filtering of the data. That data is then forwarded to the feature extraction and classification parts of the AI, which can be placed on the mobile phone by Kubernetes by providing a specific label to it.

To achieve this scenario, you first need the Cloud AI model and the embedded AI accelerator from our earlier scenario. In addition, to implement the corresponding filtering on the embedded device you need to create a new AI accelerator for on-device filtering as well as a third callback function for yet another URI on the embedded device. It is shown in Listing 7 and very similar to our embedded AI example. Again, usage of the FPGA is abstracted via a C stub into a single function call.

Listing 7 R2F Handler for embedded preprocessing

```
1 void handler_preprocess(uint16_t id){
2     uint8_t data[bufferSize];
3     uint8_t processed[bufferSize];
4     microphone_read(data, bufferSize);
5     preprocessing_execute(data,
6         processed, bufferSize);
7     r2f_response(id, processed, bufferSize)
8 }
```

The resulting embedded main function on the MCU is shown in Listing 8.

Listing 8 Initialisation and duty cycle handling for all three variants

```

1  int main(void) {
2      r2f_registerHandler(handler_rawAudio ,
3                          "EIP:// devices/X/audio");
4      r2f_registerHandler(handler_embeddedAI ,
5                          "EIP:// devices/X/ai");
6      r2f_registerHandler(handler_preprocess ,
7                          "EIP:// devices/X/pre");
8      elasticnode_init();
9      microphone_init();
10     platform_init_sleep();
11
12     while(1){
13         r2f_dutyCycle();
14         elasticnode_dutyCycle();
15         platform_sleep(SLEEP_LENGTH);
16     }
17 }

```

On the part of the embedded device, everything is now prepared to adapt your AI at runtime. For this to work, you extend the AI twin. Until now, the AI model has been a monolithic block and thus, the AI twin was a basic twin. Now, you can use an AI twin that is a composite twin. The AI is modelled as a set of different twins that are wired to each other via their URIs. Thus, e.g. preprocessing can be modelled as a twin, which internally links to the device twin to get preprocessed data. Feature extraction can be modelled as a second twin, and classification as a third one. The composite AI twin can reselect between them at runtime as it chooses, e.g. to cope for changing network connectivity. Note that this design does not require the data flow between the parts of your AI model to go through the composite AI twin. Instead, it can send to each twin the URI to which it should send data to and connect all the parts directly with each other. It only ever receives the final result of the AI model. The composite AI twin just becomes the common interface that other applications can talk to, and thus hiding the details of the AI implementation.

One final missing piece is how our composite AI twin is notified about changes in the connectivity of the embedded device. This can be done with its device twin, which provides a URI to query the current communication technology that is being used. Alternatively, when the embedded device switches its communication protocol it automatically uses a different translation service and, technically, rejoins the system. This is detected by the Bootstrapping service and our composite AI twin can be notified about it. The decision to actually adapt the AI and reselect a new configuration is currently not automated. Instead, this is done explicitly by the composite AI twin. In the future we plan to provide different decision algorithms for this, which the AI twin can call. These will be based on well-known algorithms from

related work, e.g. heuristics such as proposed in Becker et al. (2004).

Our approach for implementing adaptive AI models can be applied easily to other scenarios, too. As an example, when the context of a user changes, you might need to use a different AI model all together. Let us assume you have implemented a second AI model, consisting of another composite AI twin with the corresponding data twin, preprocessing and filtering, as well as an implemented AI accelerator for the deeply embedded device. Incorporating this new twin can be realised in two ways. First, you can develop another composite twin, which connects to both composite AI twins to switch between them. Alternatively, you can specify a new URI, which represents the combination of both AI models. When a client calls this URI, you can instruct the URI Resolver to map this to one of the URIs of the original AI models and route requests to the one that is active right now.

6 AI data adaptation

So far, we have shown how a developer can create an adaptive AI model, e.g. by updating to a new model or by redeploying it to new target devices. However, so far we have ignored our Requirement 3, namely to cope with fluctuating data source populations. Instead, we worked with a single, fixed data source. In this section, we extend our approach and show how to support dynamic populations as well as varying data qualities. To do so, similarly to Sect. 5, we discuss a sequence of increasingly complex adaptation scenarios.

6.1 Fixed data source

As a starting point for our discussion, let us first reiterate our basic case from Sect. 5.1. To get input data from a specific sensor, the developer created a simple data twin, which is connected to a fixed device twin that communicates with the device actually doing the measurements (see Listing 3 for the device twin). This approach does not allow us to reselect to new data sources. Still, it allows a very restricted parametric adaptation, which we ignored so far. The data twin provides an API to clients that allows them to specify non-functional requirements for the data, e.g. how often they need a new measurement. This allows the data twin to configure the sensor device accordingly. In this example the data twin relays the measurement frequency to be configured to the device twin of X. In Listing 9 we see the handling of this request for device twin X: It extracts the given measurement frequency (in Hz) and creates a new *TimerConfig* with it. This *TimerConfig* calculates all necessary values like the prescaler value, timer mode etc. for the embedded device, depending on the available timer hardware. Since

it is a device twin it possesses all the necessary knowledge like clock frequency, available timer hardware, of the device it's mirroring. After calculating everything it just sends the configuration to the device.

Listing 9 Simple data twin listing

```

1 DeviceTwin extends AbstractTwin{
2   @EIPMessageHandler("/setFrequency")
3   public void configAudioFreq(EIPMessage
4     message) {
5     int freq = message.parameter("freq");
6     TimerConfig config = new TimerConfig(
7       freq);
8     callURI("EIP://devices/X/freq",
9       config);
10  }
11 }
```

6.2 Reselecting a data source

In reality, binding our AI model to a single, fixed data source can lead to problems. If e.g. a user moves between rooms, a data source that is installed in the environment may become unavailable (or may not provide the required data any more). Therefore, a new data source has to be selected dynamically. As common in pervasive applications an abundance of additional data sources may now be available, e.g. a low quality sensor carried by the user and a high quality sensor installed in a room. One of these data sources has to be selected.

For this we need to (a) detect the set of sensors that measure the needed phenomenon, e.g. temperature sensors in the same room as the user, and (b) choose the most suitable sensor out of that set.

The first part is essentially a context-aware sensor discovery problem. Out of all available sensors, we need to discover the ones that have the correct type (e.g. temperature) and context (e.g. in the same room as the user). This should be done efficiently and should scale for increasing numbers of sensors. How to implement a suitable location and context management service for this has been a long standing area of research in the pervasive computing community. However, as it is not specific to implementing AI models, it is out of the scope of this paper. A context management service can be integrated into our platform as an additional twin, e.g. using our bootstrapping service and device twins to get continuous location information from all devices. We can also support multiple such services in parallel, e.g. to support different location models for different applications.

In this paper, due to the data-centric nature of AI enabled applications, we in contrast propose to detect the correct data sources based on their measured data in relation to a given reference sensor, e.g. a sensor carried by the user. Previous

work (Schmeißer and Schiele 2020) showed that sensors (for the most part) produce similar data if they measure the same phenomenon. This can be detected with an unsupervised learning algorithm for data clustering. In our current system implementation we support three such algorithms: (1) modularity based community detection (Newman 2004), (2) label propagation on a graph (Raghavan et al. 2007) and (3) modified OSort binning (Rutishauser et al. 2006; Valencia and Alimohammad 2019). A developer can choose to use one of these for their application or she can implement a different algorithm. Note that the algorithms are themselves implemented as AI models that are managed by our platform (via AI twins) just like any other AI model. Thus, they can also use the full scale of adaptation possibilities. In addition, input data for them is provided by data twins that can pre-filter by data type and network, effectively only taking into account e.g. temperature data that is measured by devices in a user's home network. We are currently exploring further possibilities to increase scalability, e.g. by combining a location-based filtering with data clustering.

Once we have found a set of sensors that measure the phenomenon that we want, we need to select the most suitable sensor out of it. This can be achieved in different ways according to the application's needs. We can use the self-description of devices to select the device with, e.g. the highest sensor accuracy, a device with the most battery power left, or simply select one randomly.

Listing 10 shows an example implementation of a data twin that chooses the most accurate temperature sensor from a set of given sensors in relation to a reference point. In this example we create our own initialisation function, which in turn uses the provided `init` and `start` function of our API. Besides the normal initialisation of a twin we use our platform's adaptation scheme to instantiate other twins to use for input. The task of the twin at hand is to pick out the most accurate temperature sensor from a given set of sensors (l. 20–26). To do so, we need the set of sensors to be provided. The set of sensors is created by one of the aforementioned algorithms, namely OSort. Thanks to our preparations, there already exists a twin that offers this algorithm as a service and is accessible via the `EIP://osort` URI. We only need to create a new configuration on it, so it takes temperature sensors as an input and performs the algorithm on it; provide our reference point to it, so it can choose the relevant set of sensors for us; and finally, provide a destination URI to which to send the relevant set of sensors (l. 11–17). To make things easier we additionally instruct the *Bootstrapping Service* to instantiate a new composite twin specifically for temperature sensors, by providing it the service description, i.e. temperature, and the base URI for the twin (l. 6–9). As a final step we offer the value of the most accurate sensor via URI (l. 28–34). Figure 9

```

sequenceDiagram
    participant G as :GroupedTempData
    participant B as :Bootstrapper
    participant O as :OSort
    participant T as :TempDataTwin

    Note over T: X

    G->>B: /new_comp_twin:  
(twins/temp_data_twin)
    activate B
    B->>T: create_new_comp_twin
    deactivate B
    T->>B: register_at_bootstrapper:  
(twins/temp_data_twin)
    deactivate T
    B->>O: /new_config:  
(temp_data_twin/data,  
/temp_data)
    activate O
    O->>T: /data
    activate T
    T-->>O: /temp_data
    deactivate T
    O-->>G: /temp_data:  
(temp_data_cluster)
    deactivate O
    G-->>G: loop
    
```

The sequence diagram illustrates the following steps:

- :GroupedTempData** sends a message `/new_comp_twin: (twins/temp_data_twin)` to **:Bootstrapper**.
- :Bootstrapper** sends a message `create_new_comp_twin` to **:TempDataTwin**. **:TempDataTwin** has a large 'X' mark above it.
- :TempDataTwin** returns a message `register_at_bootstrapper: (twins/temp_data_twin)` to **:Bootstrapper**.
- :Bootstrapper** sends a message `/new_config: (temp_data_twin/data, /temp_data)` to **:OSort**.
- :OSort** sends a message `/data` to **:TempDataTwin**.
- :TempDataTwin** returns a message `/temp_data` to **:OSort**.
- :OSort** returns a message `/temp_data: (temp_data_cluster)` to **:GroupedTempData**.
- A **loop** box encloses the final return from **:GroupedTempData**, indicating a repeated process.

6.3 Sensor fusion

Therefore, in our system the AI developer can use a *sensor fusion* approach to reduce the data from all applicable data sources to a single data item that can then be used as the input for a single AI model, regardless of the—potentially fluctuating—number of data sources. This data fusion is again provided by a data twin that can be configured for the specific type and context of the needed data. The AI twin can then be linked to this data twin directly. Our fusion algorithm can be adapted parametrically via an extended API that allows the developer to specify if she is more interested in an accurate or a more robust measurement.

```

1 GroupedTempData AbstractTwin{
2
3     public void initialise(){
4         init("temp-groupings");
5         start();
6         callURI(
7             "SYS://bootstrapper/new_comp_twin",
8             "hasURI=EIP://twins/temp_data_twin",
9             "service=has_temperature");
10
11         callURI("EIP://osort/new_config",
12             "sourceURI=
13                 EIP://twins/temp_data_twin/data",
14             "destinationURI="
15                 +this.URI+"/temp_data",
16             "recType=stream",
17             "refSensor="+this.refSensor);
18     }
19
20     @EIPMessageHandler("/temp_data")
21     public void receiveTempData
22     (EIPMessage message) {
23         this.bestSensor =
24             pickMostAccurateSensor(
25                 parseParams(message.parameters));
26     }
27
28     @EIPMessageHandler("/get_best_value")
29     public void sendValue
30     (EIPMessage message){
31         callURI(
32             message.senderURI,
33             this.bestSensor.value);
34     }
35 }

```


Note, that a developer can—once again—choose to use other, simpler algorithms for this, e.g. using an average or the mean over all relevant data, by providing her own data twin. However, since these approaches can severely degrade data quality, we do not support them out of the box.

Instead, we use a modified version of the *Fault-Tolerant Interval* (FTI) algorithm (Schmid and Schossmaier 2001). Originally designed for clock synchronisation, we adapted it for sensory measurements. The key concept behind the FTI algorithm is that no sensor is truly accurate, and as such measurements are treated as rough estimates, with a (known) degree of inaccuracy. In a nutshell, we treat an observable phenomenon, like temperature, light etc., as a random variable with a normal distribution. If a sensor reports a measurement, we assume this measurement to be the peak of a probability density function, e.g. it is the most likely value. The inaccuracy is represented by a neighbourhood with the measurement being the centre of it. The FTI now takes intersecting neighbourhoods and fuses them together, creating a singular neighbourhood. Not only does it reduce the dimensionality, but it also creates a neighbourhood that is much more meaningful than that of a single sensor. More details, as well as a comprehensive study, can be found in Schmeißer and Schiele (2020).

The output of FTI is a fused neighbourhood. To use this as input for an AI model, we have to map it to a singular value. We do this by creating a Gaussian mixture model by overlapping all original probability density functions. This calculates the probability density function of the fused neighbourhood. The peak of this function is then picked as our final output.

Using FTI increases the robustness of our system towards faulty and failing sensors, as it is similar to a majority vote. It derives its robustness from the fact that a majority of sensors agree upon a certain range. This can further increase the prediction quality by excluding sensors outlier from the overall AI model inputs, allowing a developer to define the required data quality for the model within the data twin.

Note that sensor fusion is not a way to cope with the common problem of data drift, where the input data distribution deviates from the distribution that the model was originally trained on. Instead the sensor fusion allows to exploit redundancy in the input data to gain values that are more consistent with the physical event allowing higher quality predictions.

7 Evaluation

So far we have already shown how our system fulfils our functional requirements from Sect. 2.3, namely, how to deploy on all device classes, how to update and adapt dynamically, and how to manage data sources. In this

section, we want to further evaluate our system. To show that it can be applied to different use cases, we implemented two different scenarios. The first one is a portable pervasive device that uses AI to continuously check for a user's heart problems. This use case focuses on adapting the AI model and shows how our system can be used to optimise the AI for different usage settings, e.g. with respect to energy consumption and latency. Our second use case is about developing a pervasive weather prediction service that uses embedded sensor stations that are deployed in the environment. It focuses on managing multiple data sources and how our system allows to cope with failing sensors.

7.1 Use case: portable ECG analysis

In the following section, we describe our first use case to show how an AI-based pervasive application can be developed with our system. As discussed before, our use case is to develop a pervasive device that checks for a user's heart problems. To do so, a DNN-based AI model analyses data from a portable ECG monitor. The DNN is executed in the Cloud or on the local device and is adapted depending on network availability (e.g. if the Internet is unavailable) and the device status (e.g. the battery is reaching critical levels or is recharging).

7.1.1 Implementation with the elastic AI platform

To realise this use case with our platform we implemented a number of components. Overall, the architecture resembles that of our example in Fig. 6.

We define an AI twin in the Cloud that connects to a DNN model for analysing ECG data. The AI twin receives its input from a data twin that abstracts away the source of the data, making the AI twin device agnostic. To be able to receive data from the embedded device, we also implemented a device twin, in our case representing an Elastic Node v4 that acts as the portable ECG monitor. It contains a Microchip AT90USB128 MCU and a Xilinx Spartan 7 XC7S15 FPGA. The device twin interacts with the device it represents through the Translation Service provided by the Elastic AI platform and a wireless communication channel, in this instance IEEE 802.15.4.

If the Elastic Node can communicate with its device twin, it pushes the raw ECG data periodically to the AI twin for analysis. The result is sent back to the device and—if necessary—the user is notified of a detected heart problem. If the network is unreachable, the ECG analysis task is offloaded to the FPGA, using an FPGA implementation of the analysis DNN model. The structure of our two different DNNs is shown in Fig. 10.

To provide an AI model for the Cloud we train a full precision CNN model with TensorFlow. We use full resolution

Fig. 10 CNN models for the Cloud and on the embedded device. Due to limited resources on the embedded device, it achieves a lower precision than the Cloud model

data and 1D CNNs. On the FPGA we apply a number of optimisation methods taken from Burger et al. (2020c) to produce a compressed model that can work with fewer resources. As input data, we use ECG records from the MIT-BIH Arrhythmia Database (Moody and Mark 2001) and store them on the Elastic Node's MCU to use it to simulate an ECG sensor.

7.1.2 ECG analysis accuracy

To quantify the performance of our platform, we performed a set of experiments with it. First, we want to evaluate if our approach to execute local AI on embedded FPGAs is able to handle high quality AI models for real use cases on pervasive sensor devices. For this, we measure the detection rate that our local AI can achieve. After training, our FPGA model is fairly performant, achieving 97% accuracy. Figure 11 shows the inference accuracy for each ECG label (Moody and Mark 2001), as well as the distribution of incorrect classifications. This is more than enough for typical scenarios. Nevertheless, for high risk situations, e.g. when a heart problem must be treated immediately and thus cannot be missed, bigger, Cloud-based models can achieve even higher accuracy (Chen et al. 2020; Oh et al. 2018; Yildirim 2018). Therefore, while the local FPGA-based AI model is

Fig. 11 Inference confusion matrix of the ECG analysis CNN model on a Xilinx Spartan 7 XC7S15 FPGA (Burger et al. 2020c)

Fig. 12 Latency distribution with 1000 times measurement

often powerful enough, when conditions allow it, it is preferable to use an AI model in the Cloud to achieve the highest possible detection rate.

7.1.3 Inference latency

Next, we want to compare the inference latency of the Cloud and local (e.g. FPGA-based) AI models.

Both our models process ECG records that contain 130 ECG data points for an inference. We define the latency for the Cloud inference as the time between starting to send an ECG record to the Cloud until the embedded device receives the AI result back from the Cloud AI model. This includes time for data transmission and jitter over the network. For the local inference we define the latency as the time required to perform the following five stages: (1) powering on the FPGA, (2) configuring it, (3) loading the ECG data onto the FPGA, (4) performing the inference and (5) sending the result to the MCU.

To consistently measure the latency, we repeated the inference 1000 times for both cases. Figure 12 shows the latency distribution for the Cloud and local AI model. On

average the local inference requires 95 ms, while the Cloud-based inference requires 401 ms. These results show that the latency overhead induced by our system in both settings is low. Our AI platform can therefore be used for scenarios in which fast response times are necessary.

Additionally, local inference also shows a much more stable latency. This is to be expected, since it does not suffer from any communication jitter. Note that our experiments for Cloud-based inference were conducted under ideal conditions with no packet loss on the wireless channel. With packet loss the resending mechanism would increase the latency further.

Both jitter and latency show that, if a very fast response is needed, the developer should choose local inference.

7.1.4 Adaptation latency

Performing a runtime adaptation causes overhead, most importantly with respect to latency when switching between configurations. Therefore, we briefly discuss the resulting latency in different scenarios.

Switching from local inference to Cloud inference The first scenario is to switch from a local AI model on an embedded device to an AI model in the Cloud. On the Cloud side, we deploy our model in a docker container. To switch to this model, the corresponding container must be made available and started, first. After that, the switch can be done instantaneously by the AI twin. The resulting latency therefore depends on the latency for making the container available, which in turn depends on the state of the container at the time of switching.

If the container has not been created before, it must be loaded and started by Kubernetes from scratch. Depending on the container design that can take a few milliseconds or several seconds. In our experiments, starting the container takes approx. 361 ms, which we think is a typical value for AI models.

If the container is paused for saving resources, then Kubernetes only needs to unpause it. In our experiments this takes approx. 20 ms, much faster than before. Since our system usually pauses an unused container instead of stopping it, this will be the resulting switching latency in most cases.

If the container is already active, then the AI twin only needs to redirect the data to the new AI model. The resulting latency is neglectable compared to the data transmission latency.

Switching from Cloud inference to local inference In the second scenario, the AI twin wants to switch from a Cloud-based model to a model that is executed on an embedded FPGA. To do so, the embedded FPGA must be started and reconfigured with a new configuration file to execute the correct AI accelerator. We assume that the configuration file has already been stored on the flash chip beforehand, e.g. when

Table 2 Elastic node energy consumption breakdown

	Cloud inference	Local inference
MCU	84 mW	65 mW
Wireless	70 mW	0 mW
FPGA	0 mW	123 mW
Sum	154 mW	188 mW
Energy	62 mJ	18 mJ

starting the pervasive application or when the model was used already at an earlier time. The latency of switching to a local model is the time for reconfiguring the FPGA, which is related to the size of the FPGA. For the Spartan-7 XC7S15 used in our system, this takes approx. 50 ms.

Switching models in the Cloud In the third scenario, we switch between AI models in the Cloud. With respect to latency, this case is equivalent to the first scenario and therefore leads to the same measurements. Depending on the state of the container that executes the AI model to which we want to switch (stopped, paused or running), the resulting latency in our experiments is approx. 361 ms, 20 ms, or neglectable.

Switching models on the embedded device Similarly, the latency of switching between AI models on an embedded device is equivalent to the latency of our second scenario, switching to a local model. Again, assuming that the new configuration file is already stored locally, the FPGA must simply be reconfigured, taking approx. 50 ms.

7.1.5 Energy consumption

While the latency is most certainly important for this medical application, it is also important to understand the energy efficiency of our implementation for the embedded device. To collect the necessary power measurements, we use a monitoring subsystem that is integrated on our Elastic Node platform (Burger et al. 2017; Schiele et al. 2019). The results can be seen in Table 2, comparing the power and energy consumption of the Elastic Node for the Cloud-based and local inference runs of the previous section.

We are using a CNN implementation optimized for low power consumption and very little decrease in accuracy. Still, the power consumption of local inference is slightly higher than Cloud inference, because using the FPGA has a higher power consumption than using the wireless 802.15.4 module. Since 802.15.4 is specifically designed for very low power, this is not surprising. However, it is also relatively slow. The embedded device (or at least the wireless module) has to remain active (in receiving mode) until it receives its response leading to a longer power consuming time frame compared to local inference. The length of that time frame is made up of several variables. It is largely consisting of the data transmission time, which in turn depends on the

amount of transmitted packages and their size. We send the ECG data in JSON format. To send one ECG record to the server we need to send around 986 Bytes of data to the R2F layer. Due to fragmentation at the 802.15.4 layer, the module needs to send 11 packages for one ECG record. (On our gateway side, the receiver module uses a USB-to-serial interface—which is even slower than wireless transmission—to transmit the received data to the computer.) As a result, the *energy consumption* per inference is 71.0% lower for the local AI. It can be assumed that other potential AI models with a similarly efficient FPGA implementation will produce similar results. Thus, to optimise battery life, a developer should again choose local inference when using our platform and runtime. Utilizing a different transmission technology, however, may lead to differing results. For example, a technology with a higher transmission rate will lead to a lower latency for Cloud-based AI but will presumably consume even more energy.

7.1.6 Concurrent AI tasks

As we have seen so far, our device can achieve higher speed and energy efficiency when executing an AI model on its FPGA instead of offloading it to the Cloud. We have also discussed the conditions that have lead to these results. More realistic conditions in terms of the latency experiments, including packet loss over the radio channel, would make the local inference option even more favourable in comparison. Furthermore, energy consumption for local inference is largely dependent on the energy efficiency of the AI model implementation. Its efficiency could be improved by applying more aggressive optimisations such as low-bit width quantization and pruning, although this will usually decrease the accuracy of the AI model.

However, the embedded FPGA has a major disadvantage: it can only execute one AI model at a time. Furthermore, you cannot just implement *any* AI model on our embedded FPGA. It has to be optimized so that it fits our limited resources. In conjunction with AI model decomposition and scheduling, this is ongoing research. What happens if our device needs to apply different AI models on the ECG data to detect different heart problems? This can even occur dynamically, if our AI detects an artefact that should be checked further by additional, more specialised AI models. For our runtime system, this means that multiple AI models are placed onto the same embedded FPGA and are executed sequentially using round robin scheduling. More sophisticated scheduling algorithms for this are subject to ongoing research. Additional overhead is induced by the necessity of reconfiguring the FPGA between each AI. In the Cloud, all AIs can be executed in parallel.

Based on 100 measurements for each setting, Fig. 13 visualizes the total time cost of executing multiple AI

Fig. 13 Total time cost of executing AI tasks

Fig. 14 Total energy cost of executing AI tasks

models in the Cloud or locally on the FPGA. With an increasing number of concurrent AI tasks, the total latency of local inference increases linearly, while the latency of the Cloud inference only has a slight increase. Still, despite the reconfiguration overhead, due to the large overhead of wireless transmission, local inference is still faster than Cloud inference for 6 or fewer AI tasks. In addition, according to Fig. 14, if we have less than 5 concurrent AI tasks, local inference with our Elastic Node is more energy-efficient than Cloud inference. The energy consumption overhead for switching AI models is largely dependent on the FPGA reconfiguration time and power consumption. If these parameters differ on another hardware platform, the amount of concurrent AI tasks which are feasible can change.

This leads us to the conclusion that—to achieve an optimised AI system—the developer must take into account how many AI models are currently executed that use data from a given pervasive sensing device. If the number is low, then placing the AI model on the device's embedded FPGA is both faster and more energy-efficient. However, for larger numbers of AI models using the same data, the developer should execute these models in the Cloud. This should be coordinated between all AIs, since mixing local and Cloud-based AIs may lead to suboptimal performance.

Fig. 15 Time cost per ECG record with batch processing

Fig. 16 Energy cost per ECG record with batch processing

7.1.7 Batch processing

The time and energy cost of powering on the FPGA and configuring it, has a big impact on the energy-efficiency for local inference. To counter this, we explore how batch processing can reduce the overhead of inferencing a single ECG record. To enable batch processing, we apply a buffer to temporarily store multiple ECG records before performing inference for all of them, either by sending them all to the Cloud or by feeding them all to the FPGA. For the FPGA, the overhead is reduced because we only need to activate and configure it once per batch, not for each record. For the Cloud, once the batch is received, we can distribute each record to a parallel AI instance and process them all in parallel.

Figures 15 and 16 show the latency and energy cost overhead of local and Cloud inference for batch sizes from 1 to 10 ECG records.

Although the FPGA must still process all ECG records in a batch sequentially, the overhead of local inference per record can be reduced by up to 81% for a batch size of 10 records. This shows the impact of activating and configuring the FPGA on the overall performance of the system. For Cloud-based inference, the cost can be reduced by 51%. This is due to the high energy and time cost of actually sending

the data to the Cloud compared to the time spent for the actual inference.

We conclude that, if possible, a developer should use batch processing for the AI. This can produce large savings, especially for local AI models. One scenario to exploit this is to retrain a local AI. While we may need to do inference for each new measurement right away to guarantee timely responses, we can still buffer all raw data locally for a given time frame. Then, we can reconfigure the FPGA, retrain our AI model with all buffered measurements and update our inference AI model accordingly.

7.2 Use case: pervasive weather prediction

Our second use case addresses the development of a pervasive weather prediction service. In this scenario, we focus on adapting the data sources, not on adapting the AI model, as in our first use case. Therefore, the main challenge for the AI developer is the changing number of suitable input data sources—embedded sensor stations that are placed in the environment—that can be used for different locations. In addition, it is possible that a sensor station is damaged or temporarily not functioning properly, e.g. due to birds or other animals. The AI should be able to work reliably for such cases.

7.2.1 Implementation with the elastic AI platform

We implemented a number of components to deploy our use case with our platform. Again, the architecture is similar to the one depicted in Fig. 6.

First, we created a monolithic AI twin in the Cloud that takes five different environmental phenomena into account to perform a weather forecast: wind speed, wind direction, air pressure, temperature, and humidity. The AI model is taken from Karvelis et al. (2020). For sake of simplicity we focus solely on the wind speed prediction. To reduce the development effort and the necessary training, the AI twin is designed to receive a single input value per phenomenon for each time step, independently of the number of actually available data sources. Our data sources are the aforementioned embedded sensor stations. Each one of them has a digital twin representation that regularly queries the sensor data and buffers it. To make all of their measurement data available to the AI twin, we implement two data twins. The first gathers all measurements from all data sources and provides access to all of them with a single URI call. The second data twin uses this data and applies our FTI sensor fusion algorithm to it to fuse all measurements to create a single, combined measurement per phenomenon that is then forwarded to our AI twin. As such, the AI twin is completely decoupled from the amount, URIs, and—to an extend—the quality of the sensor stations.

Fig. 17 Mean absolute percentage error for varying number of low-quality sensor stations

We compare our approach to a straight-forward alternative solution in which the AI developer applies a standard average pooling algorithm to the measurements as part of the AI's pre-processing.

Since we cannot deploy a sufficient number of real sensor stations, we use simulated weather data based on actual sensor measurements curated by the *Deutscher Wetterdienst*,¹ This also allows us to control the exact number and accuracy of sensor stations at each location to create varying settings for our evaluation. We can even let sensors fail completely to analyse the impact of faulty sensors on prediction quality.

7.2.2 Varying number of low- vs. high-quality sensors

First, we want to evaluate how well our system shields the AI model from a rising number of low-quality sensors. Our scenario includes a total of ten simulated sensor stations. We configured each sensor to have an accuracy similar to real-world off-the-shelf sensors, either for high-quality or low-quality sensors. The measurement of a sensor is sampled from a uniform distribution on an interval $[m_t - m_{\max}d, m_t + m_{\max}d]$, with m_t being the ground truth and d the sensor's maximum deviation from the ground truth relative to the maximum measurable value $m_{\max}$. For high-quality sensors we chose d from the interval $[0.005, 0.05]$. For low-quality sensors we picked from the interval $[0.01, 0.1]$ instead. Since the result of the simulation is non-deterministic, we decided to produce ten data sets per low to high quality sensor ratio. The model is trained three times on each of these data sets resulting in a total of thirty experimentations per configuration.

Figure 17 shows a boxplot of the mean absolute percentage error of the AI's weather prediction in relation to

the rising number of low-quality sensors for both average pooling and FTI. As mentioned in Sect. 6.3 our FTI-based sensor fusion algorithm provides a fine-grained control of fault-tolerance, which we abstracted into a quality-of-service (QoS) schema, with three distinct levels. Level one offers the least fault-tolerance, but highest accuracy, and level three offers the most fault-tolerance but least accuracy. In a real deployment, the developer would choose which QoS level should be used. For our evaluation we show results for all three QoS levels.

We can see that the AI's prediction error increases only slightly for FTI with up to five low-quality sensors, with the median of the mean absolute percentage error varying between 16.09 and 16.67. Average pooling on the other hand induces a large error for this setting, varying the median between 18.05 and 22.40. For more low-quality sensors, both approaches resemble each other more closely, with the median error for FTI increasing to 20.3. Note, however, that in a real world scenario, average pooling would probably perform worse, since the way that our error model induces inaccuracy may lead to the errors of multiple low-quality sensors cancelling each other out.

As another observation, in this scenario a higher QoS level does not necessarily lead to a lower mean error. This is because a higher QoS level is specifically optimised to filter out faulty sensors, not inaccurate sensors. Thus, they should only be beneficial if a sensor actually breaks and produces vastly different outputs. In such a scenario, average pooling should suffer even more, since the broken sensor will likely move the average by a large amount.

7.2.3 Sensor outliers

To examine this effect, in our second experiment we want to evaluate how well our system shields the AI from a broken sensor that has no correlation with ground truth anymore. Our scenario is similar to the one above in having ten sensors per phenomenon. We only have high-quality sensors and do not introduce low-quality sensors. However, we introduce a corrupted wind speed sensor. Starting after 600 min, it reports corrupted values for the next 600 min. After that, it recovers and returns to its normal operation. This is reminiscent of delayed maintenance on sensor stations. Figure 18 shows the result of the FTI and average pooling preprocessing for the wind speed phenomenon in relation to the ground truth. It is apparent that average pooling is greatly influenced by the broken sensor. In contrast, the values that are preprocessed by FTI on QoS level 1 are not affected by the corrupted measurements.

¹ https://opendata.dwd.de/climate_environment/CDC/.

Fig. 18 Wind speed values after applying the preprocessing algorithms (FTI with QoS level 1) to the simulated measurements vs. ground truth

Fig. 19 Root square error of the wind speed prediction based on the results of each preprocessing algorithm

To further characterise how this influences the prediction, Fig. 19 shows the root square error of the wind speed prediction with its 95% confidence interval for both approaches. As we can see, the AI reacts immediately to the inaccurate input data from average pooling, leading to a large prediction error until the faulty sensor is fixed. The AI that receives input data from FTI on the other hand is not influenced by the faulty sensor.

8 Conclusion and future work

Executing AI models in pervasive computing efficiently and effectively is not easy. Many, very heterogeneous devices must be integrated into a single runtime system that can

handle highly dynamic runtime situations. The *Elastic AI* platform provides AI developers with the means to execute their AI models in such systems. It combines deeply embedded devices with Edge and Cloud servers and uses adaptation concepts from pervasive systems to enable adaptive AI models. To mitigate the limitations of embedded devices with respect to compute power, our system allows to incorporate exchangeable HW accelerators for AI that are implemented on local embedded FPGAs.

Our platform contains three major components: (1) a Cloud and Edge runtime, providing an execution platform for AI models using Digital Twins, (2) an embedded runtime, integrating deeply embedded devices and simplifying the process to deploy AI models on them, and (3) a sensor fusion engine, providing a preprocessing component that allows to simplify the training and inference of AI models on sensor data from multiple sources.

Our evaluation shows the large potential of the Elastic AI platform to aid in the development of pervasive AI systems that can be optimised and extended for years to come after they have been deployed in the field. If energy-efficiency or short reaction times need to be prioritized, handing over execution of the AI model to an embedded FPGA can facilitate that. Even the execution of multiple concurrent AI models can benefit from that, although it is important to monitor the situation and switch to a Cloud deployment if the number of concurrent AI tasks increases and starts to induce higher latency and energy costs. The above-mentioned results apply, if low-power wireless communication is used, and if the embedded AI models are implemented with energy-efficiency and low inference latency in mind. Broken input sensors can also influence the performance of a pervasive AI dramatically. This can be mitigated by using our integrated sensor fusion, which successfully shields the AI model from such sensors.

For future work, we plan to introduce different pre-existing adaptation strategies that developers can use for automatic adaptation in common usage scenarios. We also want to equip our Elastic Node with different wireless modules to support protocols such as WiFi and Bluetooth or LoRa, so we can evaluate our approach in more use cases. In addition, we are looking into how to reduce the development effort for embedded AI models even further using code generation techniques and platform-aware neural architecture search. Finally, we want to extend our support for continuous training, including the ability to use federated learning in our system.

Declarations

Conflict of interest The authors declare that they have no conflict of interest.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

References

- Aberer, K., Hauswirth, M., Salehi, A.: The Global Sensor networks middleware for efficient and flexible deployment and interconnection of sensor networks. In: Report, Ecole Polytechnique Federale de Lausanne (EPFL) (5005), pp. 1–21 (2006)
- Amazon: machine learning on aws (2021). <https://aws.amazon.com/ai/>. Accessed 10 Mar 2021
- Aygalinc, C., Gerbert-Gaillard, E., Vega, G., Lalanda, P.: Service-oriented autonomic pervasive context. In: International Conference on Service-Oriented Computing, pp. 795–809. Springer (2016)
- Barnes, C.M., Bellman, K., Botev, J., Diaconescu, A., Esterle, L., Gruhl, C., Landauer, C., Lewis, P.R., Nelson, P.R., Stein, A., Stewart, C., Tomforde, S.: CHARIOT—towards a continuous high-level adaptive runtime integration testbed. In: Proceedings—2019 IEEE 4th International Workshops on Foundations and Applications of Self* Systems, FAS*W 2019, pp. 52–55 (2019). <https://doi.org/10.1109/FAS-W.2019.00026>
- Becker, C., Handte, M., Schiele, G., Rothermel, K.: Pcom—a component system for pervasive computing. In: Second IEEE Annual Conference on Pervasive Computing and Communications, 2004. Proceedings of the, pp. 67–76. IEEE (2004)
- Becker, C., Julien, C., Lalanda, P., Zambonelli, F.: Pervasive computing middleware: current trends and emerging challenges. CCF Trans. Pervasive Comput. Interact. **1**(1), 10–23 (2019). <https://doi.org/10.1007/s42486-019-00005-2>
- Becker, C., Schiele, G., Gubbels, H., Rothermel, K.: Base—a micro-broker-based middleware for pervasive computing. In: Proceedings of the First IEEE International Conference on Pervasive Computing and Communications, 2003. (PerCom 2003). IEEE, pp. 443–451 (2003)
- Bellman, K.L., Gruhl, C., Landauer, C., Tomforde, S.: Self-improving system integration—on a definition and characteristics of the challenge. In: 2019 IEEE 4th International Workshops on Foundations and Applications of Self* Systems (FAS* W). IEEE, pp. 1–3 (2019)
- Biswas, A., Chandrakasan, A.P.: Conv-sram: an energy-efficient sram with in-memory dot-product computation for low-power convolutional neural networks. IEEE J. Solid State Circuits **54**(1) (2018)
- Braud, T., Zhou, P., Kangasharju, J., Hui, P.: Multipath computation offloading for mobile augmented reality. In: In Proceedings of the IEEE International Conference on Pervasive Computing and Communications (PerCom 2020), Austin USA (2020)
- Breitbart, M., Schäfer, D., Edinger, J., Becker, C.: Context-aware data and task placement in edge computing environments. In: 2019 IEEE International Conference on Pervasive Computing and Communications (PerCom). IEEE, pp. 1–10 (2019)
- Brinkschulte, M., Becker, C., Krupitzer, C.: Towards a qos-aware cyber physical networking middleware architecture. In: Proceedings of the 1st International Workshop on Middleware for Lightweight, Spontaneous Environments, pp. 7–12 (2019)
- Burger, A., Cichwskyj, C., Schiele, G.: Elastic nodes for the internet of things: a middleware-based approach. In: 2017 IEEE International Conference on Autonomic Computing, ICAC 2017. IEEE, pp. 73–74 (2017). <https://doi.org/10.1109/ICAC.2017.27>
- Burger, A., Cichwskyj, C., Schmeißer, S., Schiele, G.: The elastic internet of things—a platform for self-integrating and self-adaptive IoT-systems with support for embedded adaptive hardware. Future Gener. Comput. Syst. (2020)
- Burger, A., King, D.W., Schiele, G.: Reconfigurable embedded devices using reinforcement learning to develop action-policies. In: 2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS). IEEE, pp. 232–241 (2020)
- Burger, A., Qian, C., Schiele, G., Helms, D.: An embedded CNN implementation for on-device ECG analysis. In: 2020 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops) (2020) **(to appear)**
- Caporuscio, M., Raverdy, P.G., Issarny, V.: ubisoap: a service-oriented middleware for ubiquitous networking. IEEE Trans. Serv. Comput. **5**(1), 86–98 (2010)
- Cervantes, F., Ramos, F., Gutiérrez, L.F., Occello, M., Jamont, J.P.: A new approach for the composition of adaptive pervasive systems. IEEE Syst. J. **12**(2), 1709–1721 (2017)
- Chen, C., Hua, Z., Zhang, R., Liu, G., Wen, W.: Automated arrhythmia classification based on a combination network of cnn and lstm. Biomed. Signal Process. Control **57**, 101819 (2020)
- Clemente, J., Valero, M., Li, F., Wang, C., Song, W.: Helena: Real-time contact-free monitoring of sleep activities and events around the bed. In: 2020 IEEE International Conference on Pervasive Computing and Communications (PerCom). IEEE, pp. 1–10 (2020)
- Cox, B., Galjaard, J., Ghiassi, A., Birke, R., Chen, L.Y.: Masa : Responsive multi-dnn inference on the edge. In: 2021 IEEE International Conference on Pervasive Computing and Communications (PerCom). IEEE (2021)
- Eisenhauer, M., Rosengren, P., Antolin, P.: HYDRA: a development platform for integrating wireless devices and sensors into ambient intelligence systems. In: Giusto, D., Iera, A., Morabito, G., Atzori, L. (eds.) The Internet of Things. Springer, New York, pp. 367–373 (2010)
- Ek, S., Portet, F., Lalanda, P., Vega, G.: Evaluation of federated learning aggregation algorithms: application to human activity recognition. In: Adjunct Proceedings of the 2020 ACM International Joint Conference on Pervasive and Ubiquitous Computing and Proceedings of the 2020 ACM International Symposium on Wearable Computers, pp. 638–643 (2020)
- Ek, S., Portet, F., Lalanda, P., Vega, G.: A federated learning aggregation algorithm for pervasive computing: Evaluation and comparison. In: 2021 IEEE International Conference on Pervasive Computing and Communications (PerCom). IEEE (2021)
- Elmalaki, S., Wanner, L., Srivastava, M.: Caredroid: adaptation framework for android context-aware applications. In: Proceedings of the 21st Annual International Conference on Mobile Computing and Networking, pp. 386–399 (2015)
- Embedded Systems Department UDE: Elastic Node Middleware Repository (2019). <https://github.com/es-ude/ElasticNodeMiddleware>
- Escoffier, C., Chollet, S., Lalanda, P.: Lessons learned in building pervasive platforms. In: 2014 IEEE 11th Consumer Communications and Networking Conference (CCNC). IEEE, pp. 7–12 (2014)
- Fellicious, C.: Transfer learning and organic computing for autonomous vehicles. [arXiv:1808.05443](https://arxiv.org/abs/1808.05443) (2018)

- Fukushima, Y., Miura, D., Hamatani, T., Yamaguchi, H., Higashino, T.: Microdeep: In-network deep learning by micro-sensor coordination for pervasive computing. In: 2018 IEEE International Conference on Smart Computing (SMARTCOMP). IEEE, pp. 163–170 (2018)
- Gordon, A., Chang, H., King, R.H.: Neural networks for processing data from multiple redundant sensors for mine systems management, operation, maintenance, and control. In: Sensor Fusion VI, vol. 2059. International Society for Optics and Photonics, pp. 512–521 (1993)
- Guerra, L., Zhuang, B., Reid, I., Drummond, T.: Switchable precision neural networks. [arXiv:2002.02815](https://arxiv.org/abs/2002.02815) (2020)
- Guinard, D., Trifa, V., Karnouskos, S., Spiess, P., Savio, D.: Interacting with the SOA-based internet of things: discovery, query, selection, and on-demand provisioning of web services. *IEEE Trans. Serv. Comput.* **3**(3), 223–235 (2010). <https://doi.org/10.1109/TSC.2010.3>
- Han, S., Mao, H., Dally, W.J.: Deep compression: compressing deep neural networks with pruning, trained quantization and Huffman coding. [arXiv:1510.00149](https://arxiv.org/abs/1510.00149) (2015)
- Handte, M., Schiele, G., Matjuntke, V., Becker, C., Marrón, P.J.: 3pc: system support for adaptive peer-to-peer pervasive computing. *ACM Trans. Auton. Adapt. Syst.* **7**(1), 1–19 (2012)
- Hassibi, B., Stork, D.G.: Second order derivatives for network pruning: optimal brain surgeon. In: Advances in neural information processing systems (1993)
- Herdin, C., Martin, C., Forbrig, P.: Sitadapt: an architecture for situation-aware runtime adaptation of interactive systems. In: International Conference on Human-Computer Interaction, pp. 447–455. Springer (2017)
- Houzé, É., Diaconescu, A., Dessalles, J.L., Mengay, D., Schumann, M.: A decentralized approach to explanatory artificial intelligence for autonomic systems. In: 2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C). IEEE, pp. 115–120 (2020)
- Howard, A.G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., Adam, H.: Mobilenets: efficient convolutional neural networks for mobile vision applications. *CoRR* [abs/1704.04861](https://arxiv.org/abs/1704.04861) (2017). <http://arxiv.org/abs/1704.04861>
- Iandola, F.N., Han, S., Moskewicz, M.W., Ashraf, K., Dally, W.J., Keutzer, K.: Squeezenet: Alexnet-level accuracy with 50x fewer parameters and < 0.5 mb model size. [arXiv:1602.07360](https://arxiv.org/abs/1602.07360) (2016)
- Jin, Q., Yang, L., Liao, Z.: Adabits: Neural network quantization with adaptive bit-widths. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pp. 2146–2156 (2020)
- Jouppi, N., Young, C., Patil, N., Patterson, D.: Motivation for and evaluation of the first tensor processing unit. *IEEE Micro* **38**(3) (2018)
- Karvelis, P., Mazzei, D., Biviano, M., Stylios, C.: Portweather: a lightweight onboard solution for real-time weather prediction. *Sensors* **20**(11) (2020). <https://doi.org/10.3390/s20113181>. <https://www.mdpi.com/1424-8220/20/11/3181>
- Kim, S., Kim, Y.: Toward interference-aware gpu container co-scheduling learning from application profiles. In: 2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C). IEEE, pp. 19–23 (2020)
- Konečný, J., McMahan, H.B., Ramage, D., Richtárik, P.: Federated optimization: Distributed machine learning for on-device intelligence. [arXiv:1610.02527](https://arxiv.org/abs/1610.02527) (2016)
- Kostelník, P., Sarnovský, M., Furdík, K.: The semantic middleware for networked embedded systems applied in the internet of things and services domain. *Scalable Comput.* **12**(3), 307–315 (2011). <https://doi.org/10.12694/scpe.v12i3.726>
- Krupitzer, C., Roth, F.M., VanSyckel, S., Schiele, G., Becker, C.: A survey on engineering approaches for self-adaptive systems. *Pervasive Mob Comput* **17**, 184–206 (2015). <https://doi.org/10.1016/j.pmcj.2014.09.009>
- Krupitzer, C., Sztyley, T., Edinger, J., Breitbach, M., Stuckenschmidt, H., Becker, C.: Hips do lie! a position-aware mobile fall detection system. In: 2018 IEEE International Conference on Pervasive Computing and Communications (PerCom). IEEE, pp. 1–10 (2018)
- Kuo, R.: Multi-sensor integration for on-line tool wear estimation through artificial neural networks and fuzzy neural network. *Eng Appl Artif Intell* **13**(3), 249–261 (2000)
- Lalanda, P., Gerber-Gaillard, E., Chollet, S.: Self-aware context in smart home pervasive platforms. In: 2017 IEEE International Conference on Autonomic Computing (ICAC). IEEE, pp. 119–124 (2017)
- Lalanda, P., Mertz, J., Nunes, I.: Autonomic caching management in industrial smart gateways. In: 2018 IEEE Industrial Cyber-Physical Systems (ICPS). IEEE, pp. 26–31 (2018)
- Lalanda, P., Wang, D., Vega, G., Cervantes, H., Khalid, M.A.: Service-oriented pervasive platform supporting machine learning applications in smart buildings. In: International Conference on Service-Oriented Computing, pp. 5–16. Springer (2019)
- Li, S., Walls, R.J., Xu, L., Guo, T.: Speeding up deep learning with transient servers. In: 2019 IEEE International Conference on Autonomic Computing (ICAC). IEEE, pp. 125–135 (2019)
- Lin, J., Chen, W., Lin, Y., Cohn, J., Gan, C., Han, S.: Mxnet: Tiny deep learning on iot devices. *CoRR* [arXiv:abs/2007.10319](https://arxiv.org/abs/2007.10319) (2020)
- Mahn, T., Becker, D., Al-Shatri, H., Klein, A.: A distributed algorithm for multi-stage computation offloading. In: 2018 IEEE 7th International Conference on Cloud Networking (CloudNet). IEEE, pp. 1–6 (2018)
- Mattern, F.: Pervasive/ubiquitous computing. *Informatik-Spektrum* **24**(3), 145–147 (2001)
- McDaniel, B., Teerapittayanon, S., Kung, H.: Embedded binarized neural networks. [arXiv:1709.02260](https://arxiv.org/abs/1709.02260) (2017)
- Microsoft: Azure IOT edge & AI: enabling the intelligent edge (2021). <https://azure.microsoft.com/de-de/resources/videos/build-2019-azure-iot-edge-ai-enabling-the-intelligent-edge/>. Accessed 10 Mar 2021
- Moody, G.B., Mark, R.G.: The impact of the MIT-BIH arrhythmia database (2001). <https://doi.org/10.1109/51.932724>
- Muhammad, K., Ahmad, J., Lv, Z., Bellavista, P., Yang, P., Baik, S.W.: Efficient deep CNN-based fire detection and localization in video surveillance applications. *IEEE Trans. Syst. Man. Cybern. Syst.* **49**(7), 1419–1434 (2019). <https://doi.org/10.1109/TSMC.2018.2830099>
- Musha, K., Kudoh, T., Amano, H.: Deep learning on high performance FPGA switching boards: flow-in-cloud. In: International Symposium on Applied Reconfigurable Computing, pp. 43–54. Springer (2018)
- Naber, J., Pfannemüller, M., Edinger, J., Becker, C.: Perflow: configuring the information flow in a pervasive middleware via visual scripting. In: Proceedings of the 16th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services, pp. 434–443 (2019)
- Newman, M.E.: Fast algorithm for detecting community structure in networks. *Phys. Rev. E Stat. Phys. Plasmas Fluids Relat. Interdiscip. Top.* **69**(6), 5 (2004). <https://doi.org/10.1103/PhysRevE.69.066133>
- Oh, S.L., Ng, E.Y., San Tan, R., Acharya, U.R.: Automated diagnosis of arrhythmia using combination of cnn and lstm techniques with

- variable length heart beats. *Comput. Biol. Med.* **102**, 278–287 (2018)
- Radhakrishnan, M., Misra, A., Balan, R.: W8-scope: Fine-grained, practical monitoring of weight stack-based exercises. In: 2020 IEEE International Conference on Pervasive Computing and Communications (PerCom). IEEE Computer Society, Los Alamitos, CA, USA, pp. 1–10 (2020). <https://doi.org/10.1109/PerCom45495.2020.9127379>
- Raghavan, U.N., Albert, R., Kumara, S.: Near linear time algorithm to detect community structures in large-scale networks. *Phys. Rev. E* **76**, 036106 (2007). <https://doi.org/10.1103/PhysRevE.76.036106>. <https://link.aps.org/doi/10.1103/PhysRevE.76.036106>
- Rastegari, M., Ordonez, V., Redmon, J., Farhadi, A.: Xnor-net: ImageNet classification using binary convolutional neural networks. In: European Conference on Computer Vision. Springer (2016)
- Roth, W., Schindler, G., Zöhrer, M., Pfeifenberger, L., Peharz, R., Tschitschek, S., Fröning, H., Pernkopf, F., Ghahramani, Z.: Resource-efficient neural networks for embedded systems. *arXiv:2001.03048* (2020)
- Rutishauser, U., Schuman, E.M., Mamelak, A.N.: Online detection and sorting of extracellularly recorded action potentials in human medial temporal lobe recordings, in vivo. *J. Neurosci. Methods* **154**(1), 204–224 (2006). <https://doi.org/10.1016/j.jneumeth.2005.12.033>
- Sadhu, V., Zonouz, S., Sritapan, V., Pompili, D.: Hcfcontext: Smartphone context inference via sequential history-based collaborative filtering. In: 2019 IEEE International Conference on Pervasive Computing and Communications (PerCom). IEEE, pp. 1–10 (2019)
- Saeed, A., Salim, F.D., Ozcelebi, T., Lukkien, J.: Federated self-supervised learning of multisensor representations for embedded intelligence. *IEEE Internet Things J.* **8**(2), 1030–1040 (2020)
- Sandler, M., Howard, A.G., Zhu, M., Zhmoginov, A., Chen, L.: Inverted residuals and linear bottlenecks: Mobile networks for classification, detection and segmentation. *CoRR* **abs/1801.04381** (2018). <http://arxiv.org/abs/1801.04381>
- Schiele, G., Burger, A., Cichiwskyj, C.: The elastic node: an experimentation platform for hardware accelerator research in the internet of things. In: 2019 IEEE International Conference on Autonomic Computing, ICAC 2019. IEEE, pp. 84–94 (2019). <https://doi.org/10.1109/ICAC.2019.00020>
- Schmeißer, S., Schiele, G.: Cosense: The collaborative sensing middleware for the internet-of-things. *ACM/IMS Trans. Data Sci.* **1**(4) (2020). <https://doi.org/10.1145/3395233>
- Schmid, U., Schossmaier, K.: How to reconcile fault-tolerant interval intersection with the Lipschitz condition. *Distrib. Comput.* **14**(2), 101–111 (2001). <https://doi.org/10.1007/PL00008927>
- Tantawi, A.N., Steinder, M.: Autonomic cloud placement of mixed workload: an adaptive bin packing algorithm. In: 2019 IEEE International Conference on Autonomic Computing (ICAC). IEEE, pp. 187–193 (2019)
- Turky, A., Rahaman, M.S., Shao, W., Salim, F.D., Bradbrook, D., Song, A.: Deep learning assisted memetic algorithm for shortest route problems. In: International Conference on Computational Science, pp. 109–121. Springer (2020)
- Valencia, D., Alimohammad, A.: A real-time spike sorting system using parallel osort clustering. *IEEE Trans. Biomed. Circuits Syst.* **13**(6), 1700–1713 (2019). <https://doi.org/10.1109/TBCAS.2019.2947618>
- Van Laerhoven, K., Aidoo, K.A., Lowette, S.: Real-time analysis of data from many sensors with neural networks. In: Proceedings Fifth International Symposium on Wearable Computers. IEEE, pp. 115–122 (2001)
- Venieris, S.I., Bouganis, C.S.: fpgaconvnet: A framework for mapping convolutional neural networks on fpgas. In: 2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM). IEEE, pp. 40–47 (2016)
- Venieris, S.I., Bouganis, C.S.: fpgaConvNet: a toolflow for mapping diverse convolutional neural networks on embedded FPGAs. *arXiv (Nips)* (2017). [arXiv:1711.08740](https://arxiv.org/abs/1711.08740)
- Venieris, S.I., Bouganis, C.S.: fpgaConvNet: mapping regular and irregular convolutional neural networks on FPGAs. *IEEE Trans. Neural Netw. Learn. Syst.* **30**(2), 326–342 (2019). <https://doi.org/10.1109/TNNLS.2018.2844093>. <https://ieeexplore.ieee.org/document/8401525/>
- Wang, E., Davis, J.J., Cheung, P.Y., Constantinides, G.A.: Lutnet: Rethinking inference in fpga soft logic. In: 2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM). IEEE (2019)
- Weiser, M.: The computer for the twenty-first century. *Sci. Am.* **265**(3), 94–104 (1991)
- Xilinx: Vivado design suite (2021). <https://www.xilinx.com/products/design-tools/vivado.html>. Accessed 18 May 2021
- Yang, T.J., Chen, Y.H., Sze, V.: Designing energy-efficient convolutional neural networks using energy-aware pruning. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (2017)
- Yang, Y., Wang, C., Zhou, X.: Drama: A high efficient neural network accelerator on FPGA using dynamic reconfiguration. In: Proceedings of the International Conference on Hardware/Software Code-sign and System Synthesis Companion. ACM, New York, NY, USA, pp. 1–2 (2019). <https://doi.org/10.1145/3349567.3351727>. <https://dl.acm.org/doi/10.1145/3349567.3351727>
- Yildirim, Ö.: A novel wavelet sequence based on deep bidirectional lstm network model for ecg signal classification. *Comput. Biol. Med.* **96**, 189–202 (2018)
- Yu, J., Yang, L., Xu, N., Yang, J., Huang, T.: Slimmable neural networks. *arXiv:1812.08928* (2018)
- Zhang, C., Li, P., Sun, G., Guan, Y., Xiao, B., Cong, J.: Optimizing FPGA-based accelerator design for deep convolutional neural networks. In: Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays—FPGA '15. ACM Press, New York, New York, USA, pp. 161–170 (2015). <https://doi.org/10.1145/2684746.2689060>. <http://dl.acm.org/citation.cfm?doid=2684746.2689060>
- Zhang, W., Sengupta, R., Fodero, J., Li, X.: Deep positioning: Intelligent fusion of pervasive magnetic field and wifi fingerprinting for smartphone indoor localization via deep learning. In: 2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA). IEEE, pp. 7–13 (2017)
- Zhou, B., Cheng, J., Sundholm, M., Reiss, A., Huang, W., Amft, O., Lukowicz, P.: Smart table surface: A novel approach to pervasive dining monitoring. In: 2015 IEEE International Conference on Pervasive Computing and Communications (PerCom). IEEE, pp. 155–162 (2015)

Christopher Cichiwskyj is working as a researcher and PhD student at the “Embedded Systems” group of the University Duisburg-Essen since April 2015. He received his master’s degree in “Angewandte Informatik” with a focus on distributed, reliable systems at the University Duisburg-Essen. He received his bachelor’s degree in 2013 with a special focus on pervasive computing. The core of his research interests focuses on the development of software abstractions for adaptive and reconfigurable

embedded hardware and how the reconfigurable capabilities of FPGAs can be used to deploy larger than previously possible circuit designs on embedded FPGAs.

Stephan Schmeißer worked as a researcher at the Embedded Systems group of Prof. Schiele at the University of Duisburg-Essen from 2015 to 2021. He received his bachelor’s degree in 2013 with a focus on pervasive computing. In 2015 he graduated his master’s degree in applied computer science with an emphasis on distributed and reliable systems at the University of Duisburg-Essen. His research focuses on distributed runtime environments for the Internet of Things. Particular interest is

taken in suitable abstractions to transform the Internet of Things into a seamless system of systems.

Chao Qian is working as a researcher and PhD student at the “Embedded Systems” group of the University Duisburg-Essen since April 2020. He received his bachelor’s degree in Electronic Engineering at the University of Electronic Science and Technology of China in 2015 with a focus on wireless sensor networks. From 2013 to 2017, he was working in a company designing and producing wearable devices and humanoid robots. In 2020, He received his master’s degree in “Embedded

Systems” with a focus on energy-efficient embedded AI systems at the University Duisburg-Essen. The core of his research interests are techniques to deploy artificial intelligence (AI) on reconfigurable hardware such as FPGAs. A special focus lies in the introduction of AI technologies into more embedded application domains with their limited computing resources and power budget.

Lukas Einhaus works since April 2020 as a researcher and PhD student at the Embedded Systems department of the University of Duisburg-Essen. After his Bachelors thesis about programming abstractions for concurrent embedded systems he received his Masters degree with a focus on distributed, reliable systems at the University Duisburg-Essen. In his Masters thesis he did research on quantized neural networks. His research focuses on the techniques for designing neural networks that allow for an

efficient implementation in hardware, especially on FPGAs. The main interest herein lies on so called quantized or low precision neural networks.

Christopher Ringhofer works since April 2020 as a researcher at the Embedded Systems department of the University Duisburg-Essen. He received his Bachelor’s Degree for Applied Informatics with a focus on engineering informatics in 2017 at the University of Duisburg-Essen. Afterwards he worked three years as a software engineer in a company with a focus on the Internet of Things. He received his Master’s degree in 2020. In his Master thesis he worked on the preprocessing of

ECG data and its analysis with neural networks. His current work deals with efficient neural network architectures for time-series analysis and compression on embedded systems. Ongoing research includes using hardware-aware neural architecture search to select the most efficient layer configuration and implementation when considering limited hardware resources.

Gregor Schiele is professor for embedded systems and leads the research group for embedded systems at the University Duisburg-Essen. He received his doctorate in computer science from the University of Stuttgart in 2007. His main research interest are self-organising embedded systems in the Internet of Things. A special focus is put on local embedded artificial intelligence and programming abstractions for adaptive embedded software.