Recursive design for data-driven, self-adaptive IoT services

Pantelis A. Frangoudis, Matthias Reisinger, and Schahram Dustdar
Distributed Systems Group, TU Wien, Vienna, Austria
Email: p.frangoudis@dsg.tuwien.ac.at, e1025631@student.tuwien.ac.at, dustdar@dsg.tuwien.ac.at

***Abstract*—We present a recursive approach to the design and operation of complex, data-driven IoT services, which are challenged by the traditional cloud-centric view of service delivery. To this end, we introduce a recursive IoT node abstraction, around which we sketch the architecture of a distributed runtime environment for the execution of such services, promoting self-adaptation to a first-class concept. Our framework aims to support services of arbitrary structure and complexity, dynamically configured and fluidly deployed along a heterogeneous device-to-cloud compute continuum, and operating according to provider and user intent. It features inherent privacy enhancements and aims to give users more control over their resources and data. We demonstrate how these principles can be applied to facilitate the composition, deployment and orchestration of complex machine learning workflows, which are becoming increasingly relevant for IoT applications.**

I. Introduction

Emerging data-driven IoT architectures need to accommodate large numbers of heterogeneous services that need to be composed, deployed, and autonomously and elastically scale, in order to efficiently handle voluminous data flows originating by massive numbers of IoT devices. Flexible structures need to be in place to support an ever-changing system of systems [1], [2], where devices will be dynamically composed in constructs of different complexities in order to execute diverse tasks.

Such IoT services are traditionally engineered to be cloud-centric [3]. However, the availability of advanced computational capabilities on-device or at hosts in proximity, which has given rise to the edge computing paradigm [4], [5], creates opportunities for novel decentralized service execution modes, with improved performance and increased security and privacy. These aspects are particularly important for data-driven IoT services, which often include Artificial Intelligence/Machine Learning (AI/ML) workflows [6]. With the emergence of edge computing, the next generation of the cloud is expanding beyond traditional data-center boundaries up to the IoT device space, allowing us to imagine such IoT services fluidly deployed and elastically scaled along a device-to-cloud compute continuum [7], [8].

This, however, comes with significant management and engineering challenges. Firstly, the compute continuum is heterogeneous, not only in terms of host capabilities and interfaces, but also ownership and geographic distribution. Secondly, in contrast with the cloud, resources are more volatile at the edge, while a single administrative edge domain cannot typically offer the view of an infinite pool of resources.

From a developer viewpoint, the entry barrier is high to engineer and deploy large-scale IoT systems, having to cater for the particularities of different compute infrastructures, connectivity technologies, edge platforms and IoT device providers, if autonomous operation is to be delivered over the compute continuum. A similar barrier exists from the perspective of infrastructure provision: While, if we consider the edge/fog domain as a whole, the available spare compute capacity can compete with traditional data centers [9], these resources remain largely segregated and unusable, in part due to the lack of a common framework to harness them, and in part due to the reluctance and unclear incentives of individual (micro-)providers of compute resources to share them on demand. Important security and privacy concerns are related with this last point.

To address these challenges, it is necessary to provide advanced middleware support to (i) enable the dynamic composition of services of arbitrary structure and complexity, that can self-configure and make optimal use of infrastructure resources along the compute continuum, and adapt to changing environmental conditions following user, service provider, and infrastructure owner requirements and intent; (ii) offer IoT service providers the primitives to develop and inject custom self-adaptation logic to their services, building on resource and data abstractions provided by the runtime environment; (iii) give users more control over their resources and data, offering expressive interfaces to define access policies; (iv) facilitate cross-domain operation and natively support information scoping, thus enhancing privacy.

In this direction, we introduce a *recursive IoT node abstraction*, around which we sketch the architecture of a *distributed runtime environment* that allows the composition of large-scale IoT services, with built in self-adaptive behavior at various levels, from the infrastructure up to the service level. We argue for *recursion* as a key design principle, which enables expressing complex IoT service patterns and reducing management complexity. We present the conceptual foundations of our framework (§II) and delve into the details of our architecture's functional blocks, also discussing some desirable properties of the framework (§III). To demonstrate our principles in action, we show how our recursive design can facilitate the deployment and orchestration of complex ML workflows, which are characteristic of data-driven IoT services (§IV). We then provide a short overview of related work (§V), before concluding the paper (§VI).

II. Design principles

A. Motivating examples

Our motivation for this work draws from the need to support large-scale and complex data-driven IoT services. For example, such a composite service may include training a deep learning model on remote private data on-device via federated learning [10], potentially aggregating training devices and creating a hierarchical structure [11], [12]. The output of this process (a trained deep learning model) can be served over a compute hierarchy from the device to the cloud, to provide low-latency inference. Ideally, and depending on the service, self-adaptation logic should be provided at various levels, including optimal assignment of training tasks over (groups of) remote IoT devices, and dynamic orchestration of inference serving, based on the changing conditions of the operating environment, user preferences and service/infrastructure-level objectives. Complex service structures like this should be naturally supported, and this is very challenging. In Section IV, we show how our approach can facilitate such a service.

As another example, in prior work [13] we proposed a large-scale IoT monitoring architecture that is designed around recursive principles: A *monitor* is an entity responsible for receiving streams of events and verifying temporal logic properties on them at runtime, e.g., related with an environmental phenomenon or the state of a smart-city service (such as smart parking). A monitor's output can be sent as input to another monitor with the same interfaces but different internal functionality (verifying different properties). By chaining monitors, arbitrary monitoring hierarchies can be constructed recursively depending on the service in question. While currently a monitor's physical location and allocated compute resources are static, a runtime environment that would allow the system to seamlessly self-reconfigure, e.g., by scaling up monitor resources or migrating a monitor to another more capable host or cluster, could help this service better cope with demand dynamics. This is another example that demonstrates the need for intelligent runtimes to enable next-generation IoT services.

B. Conceptual view

From a conceptual architecture perspective, we argue for *recursion* as a key design feature of future IoT services. This stems from the following observations:

- A recursive design reduces management complexity and makes it more straightforward to express and compose complex IoT service structures, when specific functionality repeats itself at different levels of abstraction. This functionality may be expressed via common primitives that can be applied recursively.
- It can facilitate the design of resource abstraction, aggregation mechanisms, and scoped data access. This is particularly important both for dealing with complexity and for enhancing security and privacy: access to data flows originating from a device can be restricted to the device's scope, and specific filters can be applied as these flows cross domain (scope) boundaries, transforming them according to privacy policies or performing aggregation operations according to specific service logic.
- It can express various system-of-systems patterns, from simple hierarchies to meshes of connected entities of varying internal complexities. For example, a device-to-cloud component hierarchy can be expressed in a recursive way, where a node could represent a single physical service entity, or a group thereof that appears as a single logical entity; both such entities could expose the same service endpoints.

These principles materialize in a *recursive node abstraction*. The complexity and form factor of such a node can vary, ranging from pico-scale devices to complex IoT ecosystems integrating other lower-complexity nodes under different and evolving topologies and configurations, as shown in Fig. 1. Each abstraction layer exposes a (subset of a) common set of primitives that enable the dynamic composition of recursive IoT structures in an elastic way and for life-spans that range widely, depending on the role and task that the structure is expected to perform.

Such a design has the potential to support complex IoT services, but this complexity comes with non-trivial management challenges. Zero-touch service management and orchestration schemes are mandated, thus promoting self-adaptation to a first-class concept.

A recursive node is as a logical entity, and its physical realization encapsulates infrastructure resources and the service components that execute on top of them, which may be part of different services owned by different tenants. At the host level, appropriate middleware support needs to be in place. We advocate for a *distributed node runtime* environment to enable recursive IoT services, which should have the following features.

Design-time developer facilities: It should offer developers expressive northbound interfaces to build self-adaptive IoT applications, hiding the complexity of the underlying system, allowing to take advantage of the native intelligence provided by the system (e.g., for resource management), and giving one single entry point to implement custom self-adaptive behavior, when desired. When the service is physically deployed, the node runtime takes care of executing this adaptation logic.

Technology integration: The node runtime should provide a technology integration layer by means of an extensible framework to build southbound communication modules. Current IoT devices and edge computers that can host service components have notoriously heterogeneous computation and connectivity capabilities (very often multiple ones on the same device), and the node runtime should be able to interface with and exploit them.

Policy expression languages: Powerful interfaces need to be in place so that users of the IoT service (or owners of the infrastructure where service components are hosted) can express requirements and policies, mainly with respect to how their resources and data can be used and exposed.

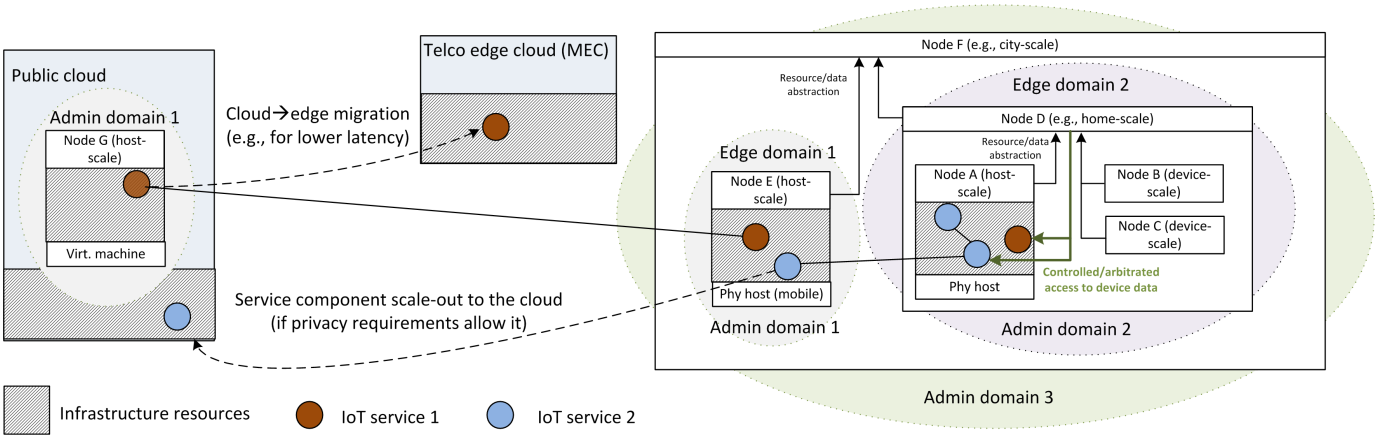


Fig. 1: Conceptual view. Nodes are recursive entities that range widely in complexity. They expose resource and data abstractions and are used for the deployment of diverse services on top of resources that span the cloud-fog-edge continuum and across administrative domains. Intelligent, self-adaptive behavior allows optimal service reconfiguration, including scaling and migration. Advanced control over data and resources is provided, to address security and privacy requirements.

Requirement translation: An adaptation layer should be offered, whose main role is to translate user and developer/service provider requirements into policies that are dynamically enforced by the node runtime, allowing adaptations to attain security, privacy and performance goals.

In the following section, we translate the above concepts and properties to a recursive architectural framework.

III. Architectural framework

A. Desirable properties

1) Requirements-driven self-adaptive behavior: Nodes are expected to host services with diverse requirements, which the node runtime should support in a unified way. A means to enable the IoT service provider to encode specific requirements and policies in the application code is via Domain-Specific Languages (DSL). The node runtime is then responsible for implementing these policies and ensuring that the requirements are met. AI-driven adaptation and policy enforcement modules running in the IoT node should operate a control loop where feedback from the node's environment is utilized for the node to decide on appropriate reconfiguration actions, in the decision space that is defined for it by the respective policy applied. Notably, the application developer should be able to implement and deploy custom self-learning behavior in a pluggable manner; the system should provide the necessary software architecture support for that. In order for the specific adaptation algorithms to collect the necessary input and derive reconfiguration decisions, node primitives for information access and exchange should be provided by the architecture.

2) AI-driven predictive behavior: A desirable property for zero-touch operation is that a node is able to learn by interacting with its operating environment. In a volatile such environment, the node should be able to deal efficiently with unknown contexts. Such intelligence pertains to different management dimensions: (i) service and (ii) resource management and orchestration. The first dimension refers to aspects related with service composition and intelligent management, for which we advocate for predictive, self-learning schemes targeting aspects ranging from node recruitment to end-to-end IoT service lifecycle management and workload placement, aiming to optimally address tradeoffs that emerge when attempting to satisfy potentially conflicting user and service provider requirements/constraints (as specified at design time) and objectives. Along the second dimension, the underlying network and compute node resources need to be orchestrated optimally to serve heterogeneous coexisting IoT application instances. Predictive models of service performance from multi-level monitoring data would be useful for optimally configuring the underlying resources to reach target service-level KPIs and node operational goals. For the latter, it is critical that the node runtime environment is capable of interfacing with heterogeneous network and compute infrastructures (see below).

3) Flexible resource allocation exploiting network softwarization: Our envisioned service architecture will be deployed across heterogeneous network environments, but mobile/wireless IoT device connectivity will be the norm. Fortunately, with the rise of network softwarization that powers the 5th generation of mobile communication systems (5G), it will be possible to dynamically and flexibly allocate customized network slices on demand [14], while the emergence of standardized interfaces for Multi-access Edge Computing [15] simplifies access to edge/fog compute resources. These developments create the technical foundation to build sophisticated resource allocation and configuration mechanisms on top. Indeed, there is a significant body of works that explores how these technologies can support IoT services [16], [17], [18].

Given a specific space of potential adaptation actions at runtime, and when the context changes or a policy violation is predicted, the node executing a service component will have to select an appropriate resource (re)allocation and reconfiguration decision. This includes (i) migrating application components across compute hosts, and (ii) allocating the

necessary network resources to maintain application performance requirements. Such actions are depicted in Fig. 1. For example, a battery-powered IoT device may be executing an AI pipeline including the following tasks: ① collect data from the environment; ② pre-process; ③ train; ④ validate; ⑤ deploy and use. In this pipeline, the training task is the most compute-intensive, while it should be possible to access the trained model with low latency. If the battery level drops below a threshold defined in the service policy, the node may decide to migrate parts of the pipeline (including the resource-hungry training task) to a nearby MEC host. Furthermore, current memory and storage constraints might not allow the model to reside on the device itself, while at the same time it should be possible for the device to use/query it at runtime with very low latency.

In situations like this, and putting this example in the context of recent ETSI standards, the node runtime can take the following decisions: (i) select an appropriate nearby MEC host, (ii) package the application components to migrate (e.g., training, validation, and model, together with an API front-end so that the device can access it) in a way appropriate for MEC deployment (as per the ETSI MEC 010-2 specifications [19]), (iii) onboard and instantiate these components as (one or more) MEC applications via the ETSI MEC Mm1 [19] and Mx2 [20] reference points, and (iv) access the 5G operator's Communication Service Management Function (CSMF) [21] API endpoint to request the deployment of an Ultra Reliable and Low Latency Communication (URLLC) network slice for the communication of the end device with the MEC application with low delay, security and reliability. Note that the node runtime might select generic (non-MEC) virtualization infrastructures to migrate the service components. In this case, the ETSI NFV-MANO Os-Ma-Nfvo interface [22] or the proprietary interface of an edge cloud provider will be used instead, via the appropriate plugin (see Section III-B).

4) Facilitating resource and data sharing (and monetization): Advanced and abundant compute and sensing capabilities are available at the edge, on top of IoT devices, and in the premises of individuals (e.g., home space) and organizations, but harnessing them for service provision in an automated and opportunistic way is challenging. At the technical level, this owes to infrastructure heterogeneity, and the lack of widespread frameworks for opening up the individual resources of a multitude of "micro-providers" in a controlled, secure and on-demand way. In addition, security and privacy concerns prevail, and the incentives to provide one's spare edge compute resources for executing tasks of other tenants are unclear [9]. At the same time, interest in putting more control on users with respect to exploiting the data they generate is growing [23].

A interesting design challenge and potential goal of IoT node middleware is thus to provide the means to individuals, collectives, and organizations to monetize on their data and resources in a (user-)controlled and flexible way, making their spare resources available on demand for the execution of IoT service components of other tenants for profit. Data generated at the edge can also be scoped appropriately for reasons of privacy and be made available to interested third-party consumers, with data providers receiving compensation.

B. Design of a distributed node runtime

At a high level, a node is composed of a set of logical building blocks. It should be noted that given the recursive nature of a node, this functionality of each block is available at any level of abstraction (from a small-scale end device to more complex structures) and exposes the same set of primitive functions. A call to a function of a specific node module may invoke recursive calls to the same function at the nodes that it contains. As a simple example, an API call to retrieve a node's available compute resource capacity will result in recursive queries to the lower-order nodes that compose it. Note that at each level of recursion, specific transformations may apply to the data returned by a function call, as a result of information scoping or other policies (e.g., anonymization operations). Fig. 2 presents the envisioned high-level view of the our node architecture, including its logical blocks and the interactions among them. The role and envisioned functionality of these blocks are sketched next.

User Facing Module (UFM): This component aims at node operators (e.g., individual users, organizations) to control access to the data their nodes generate and the use of their resources. It is accessed when an operator deploys specific third party IoT software developed with support for the node runtime. The developer (or IoT vertical) annotates at design time specific data flows for which the user needs to consent to share, and the user is presented with options to control access and visibility of this information at deployment and run time. The node operator's input is then translated by the interface module to requirements that are injected in the software. Moreover, the node operator may define pricing related options for access to its data and resources. These are communicated to the Policy Execution Engine (PEE) and propagate to the Capability Exposure Module (CEM) to expose them so that they are used in the node recruitment process, as well as to the node's Service Negotiation Module (SNM).

Capability Exposure Module (CEM): The CEM is used by nodes to advertise their resources and capabilities (e.g., sensing, hardware acceleration elements, trusted computing technology, other specific services) and provide API endpoints for third-party IoT services to use them. Depending on the complexity of the node, the CEM will provide an aggregate, abstracted view of node resources. This view is acquired by recursively accessing the CEM endpoints of constituent lower-order nodes. Furthermore, the CEM communicates with the CEMs of other nodes to discover available resources and services. The CEM implements access control policies for node data and resources according to the settings configured by the user via the UFM and propagated by the PEE. As the component in charge of advertising a node's resources and capabilities, it is also the central point where the node's context is managed and exposed via a node-local distribution broker.

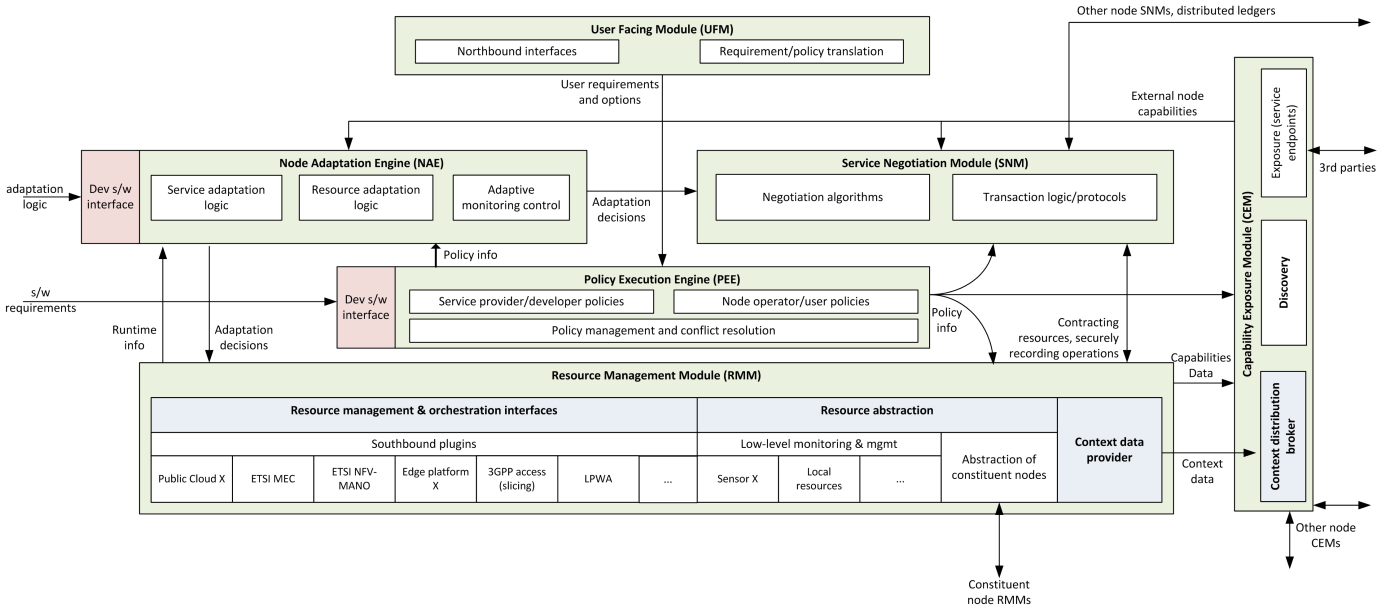


Fig. 2: Node architecture.

Service Negotiation Module (SNM): This element is responsible for executing the node's transaction logic during service composition and runtime. It executes transaction protocols for acquiring or contributing data or resources, considering policy restrictions posed by the node operator, service requirements and node capabilities, and pricing related options. Service composition algorithms are executed by the SNM, for which the latter accesses the node's CEM and invokes its discovery functions to spot other nodes' available resources and services.

Node Adaptation Engine (NAE): The NAE implements service-specific and node-wide adaptations. Much of the intelligence of the system is implemented here. The NAE executes autonomous control loops, receiving feedback from the environment and self-adapting as required by the IoT service at design time to attain application-specific goals (e.g., end-to-end latency), and by the node operator to optimize for its individual operational objectives (e.g., save on energy consumption and bandwidth cost, maximize profit from data monetization, apply security-enhancing configurations, etc.). An implementation of our framework should provide the runtime support for executing custom intelligent behavior in a pluggable manner. The NAE is responsible for executing self-adaptive algorithms for service management and orchestration, making resource allocation and orchestration decisions that are passed on to the Resource Management Module (RMM), as well as deriving adaptation decisions with respect to how service and resource monitoring should be performed to attain specific objectives (e.g., overhead reduction, energy savings). The NAE interacts with the node environment (via the RMM and the CEM) and with the PEE (that stores the translated node operator and service provider policies), receiving the necessary input via well-defined interfaces and primitives, in order to select the appropriate action from the space defined by the PEE.

Policy Execution Engine (PEE): The PEE is responsible for managing end enforcing application-specific and operator-defined rules. This also involves resolving conflicts between user and service-provider policies, and providing input to the NAE to implement policy-driven adaptation decisions. Furthermore, the PEE is responsible for propagating data access control rules to the CEM to enforce appropriate data visibility constraints, as well as to push user options and preferences (e.g., with respect to pricing) to the SNM.

Resource Management Module (RMM): This element is aware of the available compute and network resources, current workload, sensing and other capabilities of the node, as well as of their status. It implements the node runtime's layer that enables (i) resource monitoring, (ii) device resource abstractions, and (iii) interfacing with the underlying hardware, cloud-fog-edge virtualization infrastructure and communication substrate. The RMM also acts as a context data provider (and aggregator, in case a node is recursively composed by lower-order ones). The RMM arbitrates access to the data that are generated by the device hardware across the multiple services deployed over a node and managed by different tenants. It also exposes an interface to the PEE for the latter to apply developer and node operator policies (e.g., regarding data visibility) which are enforced by the node runtime. To deal with the heterogeneous underlying network and compute infrastructures across the device-to-cloud continuum and provide flexibility towards supporting future technologies, the RMM should provide an extensible southbound plugin framework. The role of this framework is to abstract their technical details and to provide a unified interface to other runtime components. Southbound plugins may include support for virtualization schemes and communication technologies such

as ETSI MEC-compliant edge compute infrastructures, public clouds, network slice orchestrators, and other.

IV. Enabling edge intelligence: Composition, distribution, and orchestration of ML workflows at the edge

A. Overview

In this section, we demonstrate the principles of our framework in action, applying our recursive design to facilitate the full lifecycle of a complex data-driven IoT service. Our aim is to showcase the versatility of such a design to support emergent IoT services, tackling challenges such as infrastructure heterogeneity, service complexity, and self-adaptive operation. Our work can be seen as an enabler for edge intelligence, and can serve its two dimensions as elaborated by Deng et. al [24]:

- *Intelligence-enabled Edge Computing (IEC)*, otherwise referred to as *AI for edge*, where the underlying edge infrastructure and the IoT services on top of it are intelligently orchestrated by means of AI.
- *AI on Edge (AIE)*, namely training and inference with device-edge-cloud synergy.

In more practical terms, we elaborate on the role and functionality of the components of our node architecture, and how they are used to compose and manage the lifecycle of a full ML workflow on top of compute infrastructure along the device-to-cloud continuum.

At a high level, such a workflow, which is typical of many data-driven applications, alternates among the following processes: (i) training a ML model on data generated at IoT devices, after appropriately pre-processing them; (ii) deploying the derived model and using it for inference; (iii) re-training, updating and re-deploying the model, either periodically or when deemed necessary.

The traditional, cloud-centric way of managing such a workflow would involve transmitting device data to centralized cloud hosts, where the compute-intensive training and validation phases take place, and then serving the model from the cloud via API endpoints; both training data and inference queries need to traverse all the way to the cloud. Following the classification of Rausch and Dustdar [25], this workflow is referred to as *Cloud to Cloud (C2C)*. Despite some advantages, C2C workflows are challenged in the following ways: (i) they do not exploit compute resources at the edge, which are recently becoming more capable with the introduction of AI accelerator hardware; (ii) model serving from the cloud comes with latencies that are often unacceptable for delay-sensitive applications typical in industrial and vehicular IoT, but also for user-facing IoT applications that feature a lot of interaction [26]; (iii) massive volumes of IoT traffic towards the cloud strain the network transport infrastructure; last but not least, (iv) stringent privacy constraints stemming from operation on sensitive data may prohibit their centralized and cross-domain processing – for various applications such as healthcare-related, private data have to be processed in place or within administrative boundaries. Therefore, our aim is to enable more diverse workflow structures, from *Edge to Edge (E2E)* ones, where all AI functionality is pushed to the edge, to services of varying structure and complexity spanning the device-to-cloud continuum (e.g., supporting hierarchical federated learning [11], [12] and distributing Deep Neural Networks (DNN) along the compute continuum [27]).

Such a complex distributed ML pipeline is shown in Fig. 3, where the role of the functional components of our framework is also put in context. This workflow is composed of two main services (themselves broken down, in turn, to a number of internal ones), distributed along the compute continuum and executed alternately in a loop that is controlled based on feedback from the service- and infrastructure-level operating environment and policies that are put in place by the service provider and node operators: (i) a federated learning (FL) process [10], where a model (a DNN) is trained on data streams originating from IoT devices, organized in a node hierarchy, and (ii) a deployment process, which receives the trained model and optimally distributes it layer-wise across a compute hierarchy that spans devices, edge nodes, and the cloud. This enables to distribute inference computation load, and, if the model allows it [28], [27], to provide inference at the edge by early exits when confidence is high; this reduces latency and communication load. Training is continuous and operates in rounds or can be triggered when performance drops. As soon as an updated model is available, it is redeployed. Redeployment can also take place when changes in the underlying compute and communication substrate are detected, warranting adaptation. In the context of our recursive design framework, these two services operate as follows.

B. Distributed training

1) Service overview: Training follows the principles of federated learning [10], which is a distributed ML paradigm allowing model training on decentralized data. A FL system [29] operates in rounds, where a server manages a FL task which is executed remotely by a number of devices (workers). This task is typically about training a ML model, such as a DNN, on local device data. A FL protocol defines procedures such as device registration, worker recruitment, parameter configuration, reporting the results of local computation, etc. A FL round can be broken down in three main phases: (i) device selection, (ii) round configuration, and (iii) reporting and aggregation of the reported computation results in the single global model.

Device selection gives rise to interesting optimization problems and defines part of the adaptation space of a FL service: Often devices are unreliable [30], heterogeneous both in capabilities [31], [32] and in terms of the distribution of their data [33], are connected over bandwidth-limited or metered links [34], or have disincentives to participate [35]. These aspects need to be taken into account when selecting an appropriate subset of the available workers to participate in a training round. The respective selection criteria depend on the particularities of the use case in question. We later demonstrate an example worker recruitment scheme, but other options are

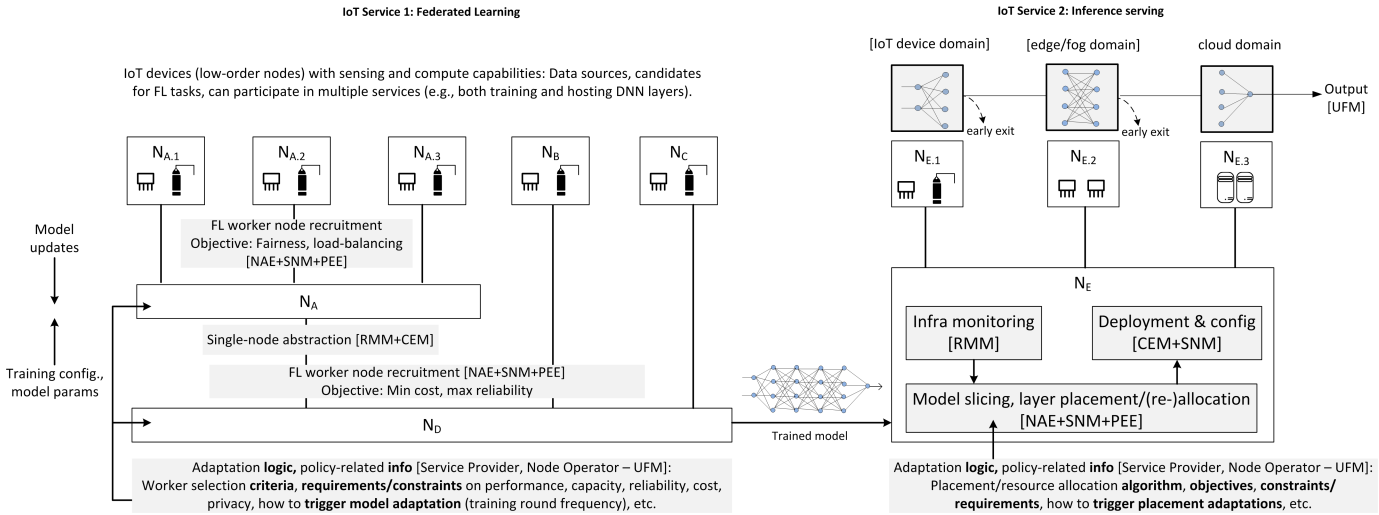


Fig. 3: A complex, recursively composed data-driven IoT service workflow spanning the device-to-cloud continuum, delivered following our concepts and framework. **LHS:** A high-order node (N_D) orchestrates a Federated Learning (FL) service. A group of low-order nodes (devices) are abstracted as a single node (N_A), which exposes the same service endpoints for participating in FL. This single-node abstraction is supported by the RMM (for resource abstraction) and the CEM (to expose node service endpoints and capabilities). Adaptation logic pertains to device recruitment and is executed by the NAE of the respective nodes at different levels of abstraction, subject to policy (PEE) and other constraints. **RHS:** N_D is "chained" with a model deployment and serving node (N_E) to compose an end-to-end ML workflow. Similar principles apply here: the trained model (DNN) is sliced layer-wise and deployed on top of heterogeneous lower-order nodes (device, edge and cloud). Self adaptation builds on service and environmental monitoring (RMM), custom adaptation logic (NAE), and infrastructure/service provider policies (e.g., quality thresholds, budget and resource usage constraints; PEE). The CEM is responsible of exposing model serving API endpoints.

possible. Notably, since the operating conditions change, the system needs to self-adapt. For example, the battery level of a device may fall below a threshold which can reduce its availability, while its connectivity conditions might change, affecting both the cost and reliability of fulfilling the FL task. These need to be properly monitored and reported as part of the FL protocol, and considered in the recruitment process.

2) Node composition: In this example, IoT devices are sources of data streams on which training takes place. Our framework allows for a hierarchical composition of nodes. A node can correspond to a single device, but also can aggregate a group of devices, thus forming a higher-order structure. Node capabilities are exposed over the interfaces provided by each node's CEM. At the lower tier of this hierarchy, each node's resources are managed by a device-local RMM. As shown in Fig. 3, a higher order node (N_A) provides a single-node abstraction for a set of IoT devices. A call to N_A's RMM, for example regarding the node's aggregate compute capabilities or its overall reliability to execute a FL task invokes recursive calls to the RMMs of IoT device-level nodes. FL server functionality is executed by the highest-order node (in this case, N_D). Thanks to the abstraction provided by our recursive framework, N_A can be seen as a single FL worker, at the same level of hierarchy as low-order nodes N_B and N_C, although internally it can execute different worker selection logic, optimizing for different criteria than N_D (e.g., load-balancing tasks across devices so as to achieve long term intra-node fairness goals). The highest order node can provide a single entry point to the FL service, also being capable of serving the learned model or providing it to another node/process for deployment and serving (N_E, in our example).

While we are not discussing such issues in depth in this work, the role of the SNM is also critical in this process, as an enabler for negotiating participation in the service. This may involve different steps depending on the ownership scenario assumed: For instance, complex accounting protocols can be put in place in a multi-domain scenario, where users wish to monetize on the use of their data. In contrast, for a service where the infrastructure and data are controlled by a single entity, service negotiation takes place solely on the grounds of the resources available by each node (i.e., nodes are cooperative and node composition considers only whether or not a device is capable of execute a given task). In both cases, it is the role of the PEE to enforce the appropriate information scoping. This also includes protecting the identity of workers from the server, if this is a concern. In this case, it is allowed to register with the aggregator node, if it is considered trusted, but not directly with the FL server; the PEE and CEM of each device-level node can control access to such information.

3) Supporting hierarchical federated learning: Our design naturally supports hierarchical federated learning. This has multiple advantages. First, it may enhance the privacy of

IoT data sources by abstracting them via higher order nodes. Second, it reduces communication load by partial model aggregation at intermediate FL servers close to or at the edge [11], and can also reduce communication latencies [12]. Finally, it reduces the complexity of managing large-scale federated learning tasks. For example, as we show in Section IV-B4, when the worker population is very large, selecting the optimal set of participants for a round is computationally expensive. Having to deal with smaller populations, delegating part of the task of recruiting workers to the edge (e.g., to node N_A in Fig. 3) saves time and server resources, while it allows for applying different selection criteria at different levels of abstraction.

4) Adaptation space – Intelligent worker selection: In this FL service, there is a significant amount of parameters that can control its operation and affect its performance. It is the responsibility of the service engineers to define these control knobs and potentially develop specific self-adaptation logic that the node runtime (NAE) will execute, using as input (i) developer and user intent, and (ii) monitoring information flows originating at each node's RMM. As an example of adaptation, we focus on the process of FL worker selection.

The provider of a FL service may apply different criteria to select workers. In general, if the necessary number of workers participating in a round fail to report in time, the round is abandoned [29]. Therefore, selecting workers that are reliable is desirable. At the same time, participating in a round incurs computation and communication cost for devices, therefore minimizing it is a goal. Node reliability and cost can be expressed in various ways. Reliability can be a function of a worker's battery level (operating on low battery can not only cause the training process to fail to complete, but also creates disincentives to contribute compute resources), connectivity profile (a device connected over an error prone link is more likely to fail to report in time), or in general the history of a worker's reporting behavior (keeping track of the number of times a recruited worker failed to deliver is straightforward). Similarly, participation cost is linked with a device's connectivity properties (LTE connections are typically metered, contrary to fixed wired broadband links) and computation costs.

Assuming that node reliability and participation cost can be abstracted by unit-less metrics, worker selection can be formulated as the following optimization problem:

$$\text{minimize} \quad \sum_{i=1}^{n} c_i x_i \tag{1}$$

$$\text{s.t.} \quad \sum_{i=1}^{n} p_i x_i \geq \mathcal{E} \tag{2}$$

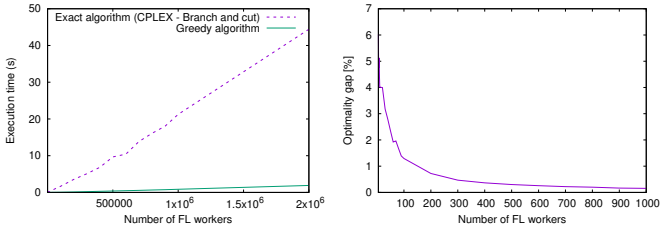
$$x_i \in \{0,1\}, \quad i = 1, \ldots, n, \tag{3}$$

where the objective is to select a subset of the n available workers (each denoted by a binary variable x_i; $x_i = 1$ if the respective worker has been selected) that minimizes the total participation cost, subject to the constraint that the expected number of delivered reports exceeds a threshold $\mathcal{E}$ (2). In this formulation, each worker i is characterized by its probability p_i to successfully deliver a report (model update) at the end of a round, and its participation cost c_i. Constraint (3) ensures that the decision variables are binary.

Successful reporting probabilities (p_i values) and costs (c_i values) need to be communicated to the FL server that carries out the selection. This functionality cuts across various functional components of our framework. The RMM of the high-order node that controls worker selection (e.g., N_D in Fig. 3) needs to communicate with the RMMs of its constituent nodes (N_A, N_B, and N_C), in order to retrieve information about the status of their resources (e.g., battery level, available bandwidth) or indications about their reliability levels, and derive the p_i values. This may be translated to recursive monitoring requests, as in the case of N_A. In a similar fashion, and depending on the implementation of the service in question, each node's CEM can advertise node capabilities, FL protocol service endpoints, and participation costs (which may be configured by node operators). The actual worker selection is functionality pertaining to the NAE of the node hosting a FL server. Finally, based on the solution to the selection problem, the SNMs of workers and the FL servers may be involved, for instance to execute transaction logic and the respective accounting procedures.

For very large-scale FL tasks, where workers are in the order of millions or more, deriving optimal solutions to problems such as the above may be time and resource consuming. To demonstrate this issue, we have used the CPLEX optimization suite to solve the worker selection problem to optimality. As Fig. 4a shows (dashed line), this can take time in the order of many tens of seconds for large problem instances, while its memory requirements become prohibitive (it was not possible to solve problem instances of more than 2M candidate workers on an 8CPU VM with 8 GB RAM due to lack of memory). Instead, in such cases, it is necessary to trade optimality for execution speed and lower resource consumption. For example, an algorithm that sorts workers in ascending order of the ratio $\frac{c_i}{p_i}$ (i.e, low-cost and high-reliability nodes first) and greedily selects workers in that order, until the expected number of delivered reports is achieved, executes much faster (Fig. 4a, solid line) while not suffering significantly in terms of solution quality in our experiments (Fig. 4b; for each worker, reliability and cost were drawn uniformly at random from the $(0.5, 1)$ and $(0, 1)$ ranges, respectively). It is important to point out that a recursive service design helps in dealing with this complexity: by using higher-order edge nodes to abstract groups of IoT devices, the central FL server node has less worker state to manage, and deals with worker selection problem instances of smaller size, delegating some of the selection logic.

In practice, other approaches can also be applied for worker selection. For example, balancing the load across workers to avoid selection bias should be considered. While our example captures critical properties and constraints, more sophisticated selection mechanisms should be expected, and other criteria



(a) Execution time of the exact and the greedy algorithm for increasingly large sets of candidate workers.

(b) Loss in performance (solution quality) of the greedy algorithm in- compared to the exact one.

Fig. 4: Solving the worker selection problem. For very large problem instances, the computation and memory cost to solve the problem to optimality can be prohibitively large and heuristic or approximation algorithms that do not significantly sacrifice on solution quality may be necessary.

and formulations are possible.

In summary, self-adaptation may manifest itself, among others, in the following ways: (i) device-level SNMs may adapt their pricing or decide to participate in a FL round based on high-level goals (intent) such as profit maximization and energy conservation; (ii) high-order nodes may autonomously select different subsets of workers across rounds to attain goals encoded in optimization criteria and respecting specific service- and infrastructure-level constraints.

C. Inference serving

1) Service overview: Once a model is available by the training service, it is ready for serving. In our example, node N_E receives the trained model from N_D and handles internally its deployment over an underlying node hierarchy, as well as exposes service endpoints that interested entities can access for inference. Orchestrating the lifecycle of model serving can be broken down to the following processes: ① processing the model and preparing it for deployment over one or more compute nodes; ② recruiting resources over the compute continuum to optimally serve inference requests based on specific criteria and constraints; ③ deploying the model, i.e., placing sub-tasks of the inference workflow on the recruited computational elements, appropriately configuring and chaining them, and configuring API endpoints for data ingestion and delivery of inference results; ④ continuously monitoring the service and the infrastructure, and, when deemed appropriate, applying adaptations to attain operational objectives such as inference latency and resource use; ⑤ terminating the service and decommissioning resources, e.g. upon the request of the service provider.

The traditional way of executing this workflow is to allocate cloud resources and host the whole ML model there. The purely edge-centric approach involves hosting the model either on-device, or offloading it to edge servers in proximity. Our approach is to allow exploiting both edge and cloud resources, and we do so by providing the appropriate execution environment. This allows a DNN model to be sliced and optimally deployed over the nodes that support hosting it. Distributing the computations of a neural network structure across nodes by model splitting and edge offloading is a topic recently receiving attention [27], [36], [35], [37], [38], [39], [40].

At runtime, the status of the network and compute infrastructure, as well as the performance of the service, are monitored and adaptation decisions are triggered when necessary (as decided by each node's NAE). Given the adaptation space defined by service and infrastructure providers, the goal of the system is to enable the fluid management of service components, e.g., facilitating dynamic layer migration across hosts when drops in network performance are predicted or when node resources are about to get exchausted, up/down-scaling the allocated compute resources following service demand (inference requests), etc. AI-driven mechanisms can be applied by the NAE towards this end, thus showcasing *intelligence-enabled edge computing* [24]. We demonstrate aspects of self-adaptation in Section IV-C3.

2) Node composition: In the same spirit as for the FL service, these nodes can be recursively composed. As Fig. 3 (RHS) shows, N_E has an overall view of nodes in the three compute domains, i.e., the device, the edge/fog, and the cloud. Although the figure does not demonstrate it, each domain can be abstracted by a single node, or individual device-level nodes can be exposed. Each node, via its CEM, exposes its capabilities to host parts of the DNN model (e.g., the service endpoints to receive computation tasks). These capabilities are combined with up-to-date monitoring information maintained by the RMM of the node on top of the hierarchy (N_E) and are utilized by its NAE when deciding on how to split, deploy, configure, and serve the model. After a DNN model is distributed over the underlying compute infrastructure, which spans the device, edge, and cloud domains, the CEMs of specific nodes that can receive input data and serve the model are configured.

Note that in Fig. 3, the input layer of the DNN model is placed at device-level node $N_{E.1}$, which is the source of input data (e.g., sensor readings). It might as well be the case that such a node also participates in the training service. This is a manifestation of a desired feature for our framework, namely to support multi-service nodes; taking this a step further, this is also a demonstration of service multi-tenancy, where a node's resources can be shared by multiple IoT services, potentially managed by different providers. It is the responsibility of each node-local PEE to decide on a resource usage policy, which is then enforced at the low level by the node's RMM.

3) Adaptation space – Layer placement, monitoring, and reconfiguration: We have designed and implemented an orchestration scheme for distributed DNN serving, which showcases various aspects of our framework design. It combines logic that pertains to all the functional blocks of our framework, and we demonstrate here how it facilitates the self-adaptive management of a model serving workflow.

Model partitioning: We provide a programming model building on top of PyTorch which allows developers to compose and train neural network architectures, potentially with early

exits, and define distributable layers. A DNN produced this way can be passed on by the service provider to a controller, which implements the core of our orchestration logic (part of the NAE) and which takes care of the distribution of its layers to compute hosts. More importantly, we also provide an automated model slicing mechanism that operates on a pre-trained, vanilla PyTorch model (in particular, on TorchScript,[1] an intermediate representation of serialized PyTorch modules), scans its computational graph, and automatically identifies split points. The processed model can be submitted for serving and is treated by the controller in an identical manner as the ones built using our developer facilities. Therefore, our system can orchestrate existing, pre-trained models, without modifications.

Host execution environment: Each compute node implements an environment that allows to receive DNN layers, perform the respective computations, communicate with other nodes in the compute hierarchy, trigger inference, and perform monitoring. Communication tasks are carried out over gRPC. This includes exchanging serialized intermediate results of DNN inference between layers. Each node's RMM is responsible for monitoring various environmental conditions which are relevant for orchestration. For example, periodically, each node measures the latency to reach other known hosts, as well as the available bandwidth. This information is also updated in the course of inference, when the node has to exchange information with nodes hosting other DNN layers. The controller (see below) is responsible for collecting, via its own RMM, and maintaining this information centrally. Nodes initially register with the controller, and the latter is responsible for contacting them periodically (CEM functionality) to determine their availability.

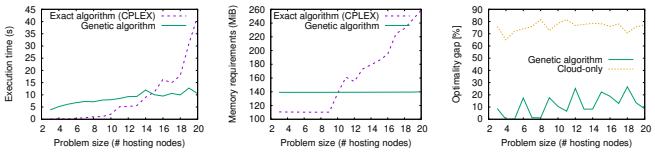
Controller engine: At the heart of our system is a controller, which implements the core NAE-level logic. The RMM of the controller maintains an up-to-date view of system-wide information. For example, it maintains a latency and bandwidth matrix with the respective metrics for any pair of compute nodes, which it updates by accessing the relevant information of the RMMs of the lower-order nodes. It exposes both southbound and user-facing APIs; the latter are used as an entry point to deploy a DNN model. From a software architecture perspective, we provide a plugin framework for implementing different deployment, resource allocation, and adaptation strategies. This way, different optimization goals can be supported. Our system readily supports a number of them, which we review here. Notably, the controller also provides an extensible configuration interface, via which various operational settings and constraints can be expressed.

Layer placement strategies: When a model is provided to the controller for serving, the latter needs to decide on an appropriate placement of layers over nodes in a compute hierarchy. This decision can be driven by different criteria. We briefly describe latency-driven strategies, but other ones are supported, such as those aiming to balance inference load fairly across nodes or minimize energy consumption. We define inference latency as the end-to-end latency from submitting input data to receiving an inference result. This is broken down to communication- and computation-related latency. Communication latency is determined by the underlying network links and the volume of data that need to be exchanged between nodes hosting different DNN layers. Computation latency is a function of the computational requirements of each inference task, and the capabilities and load of the processor executing it. The purpose of a latency-driven strategy is to determine the appropriate DNN split points and deploy layers of the DNN on hosts to minimize inference latency, subject to capacity (node and network link) and layer ordering constraints. In order to acquire the necessary input to the algorithm, we combine the following information: (i) the computation requirements per layer (in FLOPS) and the incurred traffic volume, for which we have implemented DNN profiling facilities, (iii) the computation capabilities (in FLOPS) of different compute node hardware models, which we acquire either via existing studies [41] or via offline benchmarking (for new devices), (iv) monitoring information from the RMM, and (v) the architecture of the DNN to deploy.

When the number of potential host nodes is small, or when device-level nodes are abstracted behind higher-order ones, thus limiting the search space, this problem can be solved to optimality fast. However, when deploying very deep neural architectures over large-scale infrastructures with many user-controlled device-level nodes, and deep fog computing hierarchies, deriving the optimal DNN layer distribution is computationally expensive. We formulate different variations of the problem as an Integer Linear Program (ILP), and implement alternative algorithms to solve it, which trade solution quality for execution speed. (We omit its formulation due to space limitations.) As Fig. 5 shows, as the number of candidate compute nodes grows, we can witness a steep increase in computation and memory requirements to find the optimal deployment of GoogLeNet [42], a 22 layers deep Convolutional Neural Network via an exact algorithm (implemented in CPLEX). Heuristics, such as a genetic algorithm we have implemented, even without significant tuning and domain-specific optimizations, help deal with large problem instances, without significantly sacrificing on solution quality (see Fig. 5c). The expected latency benefits compared to a cloud-only deployment are also obvious; unsurprisingly, by pushing the whole model to the cloud, we suffer an inference latency performance loss of up to 80%. This is an early effort; more sophisticated algorithms and an extensive evaluation are part of our ongoing work.

Triggering self-adaptation: The above results demonstrate the cost of layer placement, and are important when putting adaptations in the picture. The NAE, when triggering model re-deployment, needs fast mechanisms for fluid layer migration. Currently, re-deployment takes place in our system after periodic monitoring by the controller, which might fire adaptation triggers when the conditions change, and when an updated model is available. Future work will focus on

[1] https://pytorch.org/docs/stable/jit.html



(a) Execution time of the exact and the genetic algorithm for increasingly large sets of compute nodes.

(b) Memory requirements of the exact and the genetic algorithm for increasingly large sets of compute nodes.

(c) Loss in performance (solution quality) compared to the optimal solution.

Fig. 5: Solving the distributed DNN deployment problem. While for small-scale deployments the problem can be solved to optimality in acceptable time, large compute hierarchies come with significant compute and memory resource requirements. A genetic algorithm (without significant tuning) executes faster for large problem instances, suffering a reasonable loss in performance. A cloud only solution comes with significant inference latency.

learning-based such mechanisms, building on Reinforcement Learning principles [43], as a further step towards better reflecting both modes of edge intelligence; EIC and AIE.

As a final note, from an architectural and methodological perspective, our approach is similar in spirit to SPINN [37], the state of the art in Distributed DNN orchestration. Unlike SPINN, which partitions computations between a device and a server, our approach allows for more split points along the compute continuum, and *physically* partitions the model data structure layer-wise over compute nodes, so that each one of them need not host the whole model. We support multiple decision algorithms as deployment strategy *plugins*. As such, multi-objective orchestration schemes like those of SPINN can be integrated. Finally, while we designed our system with early-exit models in mind for latency-driven use cases, we support out of the box traditional neural architectures for use cases where other objectives matter, such as distributing inference load across a number of edge devices that cannot individually accomplish it due to resource constraints (e.g., small IoT devices that cannot fit a full model in memory).

V. Related work

Recursion as a design principle: Recursion has been explicitly or implicitly considered as a principle for studying and designing systems in various disciplines, and recursive design has had various interpretations. In the field of cybernetics, Beer introduced the Viable System Model (VSM) [44], studying the fundamental viability properties of organizational structures. According to the VSM, systems that survive are evolvable and adaptable, and are characterized by a recursive structure: *"any viable system contains, and is contained in, a viable system."* Barba [45] takes a design-theoretic stance, and argues for recursive thinking as a vehicle to deconstruct and solve design problems. Shlaer and Mellor [46] use the term recursive design for their proposed software development methodology, where the focus is on the precise specification of system components and their composition in an application-agnostic architecture.

Importantly, recursion has seen use in attempts to redesign the Internet architecture. Touch et al. [47], [48] observe that basic network protocol operations are repeated at different layers of the network stack. Therefore, they introduce a Recursive Network Architecture (RNA), which reuses a single, configurable *metaprotocol* recursively across different layers of the stack, each time with layer-dependent services. Day et al. [49], [50] start from the premise that networking is inter-process communication (IPC) functionality that repeats over different scopes, and develop the Recursive Internetwork Architecture (RINA).[2] In RINA, (remote) processes communicate via a Distributed IPC Facility (DIF), the counterpart of a layer, which provides communication services (e.g, routing, transport, and management). At the same time, a single process provides such IPC services to higher-layer DIFs, leading to a recursive IPC structure. The concepts of RNA and RINA have been unified in the design of DRUID [51].

Intent-based self-adaptive middleware: Building self-adaptive software is challenging both from a developer and a system (middleware) support perspective [52], [53]. To jointly address these challenges, Proteus [54] provides a programming model that allows the developer to specify measurable parameters (feedback), adjustable variables, and intent for an application. A runtime environment that uses tools from ML and control theory adapts the application to satisfy the expressed intent. Proteus was demonstrated in an adaptive video encoding application. In the IoT space, a work worth noting is SDG-Pro [55], an everything-as-code programming framework for software-defined cloud-based IoT systems, including a programming model, and the underlying middleware and execution environment. While it does not address explicitly self-adaptation aspects and it has a different interpretation of intent (in SDG-Pro, intents map to tasks with defined scopes upon which they can take effect, rather than desired states or outcomes), it does define programming abstractions to specify intents and associated attributes (such as quality- or cost-related), which allow a level of self-optimization at runtime.

VI. Conclusion

We argued for recursion as a fundamental design principle for next-generation data-driven IoT systems and services, and provided an architectural sketch of a runtime environment to materialize our concept. Having demonstrated its potential to facilitate complex ML workflows, which are becoming increasingly common in the IoT, as well as challenging to support, our future work will focus on a full prototype implementation of our node runtime, while also concentrating on intelligent, zero-touch mechanisms for service and resource orchestration.

References

[1] A. Bondavalli *et al.*, Eds., *Cyber-Physical Systems of Systems - Foundations - A Conceptual Model and Some Derivations: The AMADEOS*

[2] https://csr.bu.edu/rina/

Legacy, ser. Lecture Notes in Computer Science. Springer, 2016, vol. 10099.

[2] G. Fortino *et al.*, "Internet of Things as system of systems: A review of methodologies, frameworks, platforms, and tools," *IEEE Trans. Syst. Man Cybern. Syst.*, vol. 51, no. 1, pp. 223–236, 2021.

[3] H. L. Truong and S. Dustdar, "Principles for engineering IoT cloud systems," *IEEE Cloud Comput.*, vol. 2, no. 2, pp. 68–76, 2015.

[4] M. Gusev and S. Dustdar, "Going back to the roots–the evolution of edge computing, an IoT perspective," *IEEE Internet Comput.*, vol. 22, no. 2, pp. 5–15, 2018.

[5] M. Satyanarayanan, "The emergence of edge computing," *Computer*, vol. 50, no. 1, pp. 30–39, 2017.

[6] F. Firouzi *et al.*, "The convergence and interplay of edge, fog, and cloud in the AI-driven Internet of Things (IoT)," *Information Systems*, p. 101840, 2021, in press.

[7] M. Villari *et al.*, "Osmosis: The osmotic computing platform for microelements in the cloud, edge, and internet of things," *Computer*, vol. 52, no. 8, pp. 14–26, 2019.

[8] J. Arulraj *et al.*, "eCloud: A vision for the evolution of the edge-cloud continuum," *Computer*, vol. 54, no. 5, pp. 24–33, 2021.

[9] H. Gedawy *et al.*, "FemtoClouds beyond the edge: The overlooked data centers," *IEEE Internet Things Mag.*, vol. 3, no. 1, pp. 44–49, 2020.

[10] P. Kairouz *et al.*, "Advances and open problems in federated learning," *Found. Trends Mach. Learn.*, vol. 14, no. 1-2, pp. 1–210, 2021.

[11] L. Liu *et al.*, "Client-edge-cloud hierarchical federated learning," in *Proc. IEEE International Conference on Communications (ICC)*, 2020.

[12] M. S. H. Abad *et al.*, "Hierarchical federated learning across heterogeneous cellular networks," in *Proc. IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2020.

[13] C. Tsigkanos *et al.*, "Edge-based runtime verification for the Internet of Things," *IEEE Transactions on Services Computing*, 2021, preprint.

[14] I. Afolabi *et al.*, "Network slicing-based customization of 5G mobile services," *IEEE Netw.*, vol. 33, no. 5, pp. 134–141, 2019.

[15] *Multi-access Edge Computing (MEC); Framework and Reference Architecture*, ETSI Group Specification MEC 003, V2.2.1, Dec. 2020.

[16] Y. Liu *et al.*, "Toward Edge Intelligence: Multiaccess Edge Computing for 5G and Internet of Things," *IEEE Internet Things J.*, vol. 7, no. 8, pp. 6722–6747, 2020.

[17] L. Nkenyereye *et al.*, "MEIX: evolving multi-access edge computing for industrial internet-of-things services," *IEEE Netw.*, vol. 35, no. 3, pp. 147–153, 2021.

[18] S. Wijethilaka and M. Liyanage, "Survey on network slicing for Internet of Things realization in 5G networks," *IEEE Commun. Surv. Tutorials*, vol. 23, no. 2, pp. 957–994, 2021.

[19] *Mobile Edge Computing (MEC); Mobile Edge Management; Part 2: Application lifecycle, rules and requirements management*, ETSI Group Specification MEC 010-02, V2.1.1, 2019.

[20] *Multi-access Edge Computing (MEC); Device application interface*, ETSI Group Specification MEC 016, V2.2.1, Apr. 2020.

[21] *Study on management and orchestration of network slicing for next generation network*, 3GPP Technical Report TR 28 801, v15.0.1, Release 15, Jan. 2018.

[22] *Network Functions Virtualisation (NFV); Management and Orchestration*, ETSI Group Specification NFV-MAN 001, V1.1.1, Dec. 2014.

[23] N. Laoutaris, "Why online services should pay you for your data? The arguments for a human-centric data economy," *IEEE Internet Comput.*, vol. 23, no. 5, pp. 29–35, 2019.

[24] S. Deng *et al.*, "Edge intelligence: The confluence of edge computing and artificial intelligence," *IEEE Internet Things J.*, vol. 7, no. 8, pp. 7457–7469, 2020.

[25] T. Rausch and S. Dustdar, "Edge intelligence: The convergence of humans, things, and AI," in *Proc. IEEE International Conference on Cloud Engineering (IC2E)*, 2019.

[26] C. Wu *et al.*, "Machine learning at facebook: Understanding inference at the edge," in *25th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019.

[27] S. Teerapittayanon *et al.*, "Distributed deep neural networks over the cloud, the edge and end devices," in *Proc. 37th IEEE International Conference on Distributed Computing Systems (ICDCS'17)*, 2017.

[28] S. Scardapane *et al.*, "Why should we add early exits to neural networks?" *Cogn. Comput.*, vol. 12, no. 5, pp. 954–966, 2020.

[29] K. A. Bonawitz *et al.*, "Towards federated learning at scale: System design," in *Proc. 2nd SysML Conference*, 2019.

[30] C. Ma *et al.*, "Federated learning with unreliable clients: Performance analysis and mechanism design," *IEEE Internet of Things Journal*, 2021, preprint.

[31] A. M. Abdelmoniem *et al.*, "On the impact of device and behavioral heterogeneity in federated learning," *CoRR*, vol. abs/2102.07500, 2021. [Online]. Available: https://arxiv.org/abs/2102.07500

[32] A. M. Abdelmoniem and M. Canini, "Towards mitigating device heterogeneity in federated learning via adaptive model quantization," in *Proc. 1st Workshop on Machine Learning and Systems (EuroMLSys '21)*, 2021, p. 96103.

[33] M. Mohri *et al.*, "Agnostic federated learning," in *Proc. 36th International Conference on Machine Learning (ICML)*, 2019.

[34] J. Konečný *et al.*, "Federated learning: Strategies for improving communication efficiency," *CoRR*, vol. abs/1610.05492, 2017. [Online]. Available: http://arxiv.org/abs/1610.05492

[35] J. Kang *et al.*, "Incentive design for efficient federated learning in mobile networks: A contract theory approach," in *Proc. IEEE VTS Asia Pacific Wireless Communications Symposium (APWCS)*, 2019.

[36] C. Hu *et al.*, "Dynamic adaptive DNN surgery for inference acceleration on the edge," in *Proc. IEEE INFOCOM*, 2019.

[37] S. Laskaridis *et al.*, "SPINN: synergistic progressive inference of neural networks over device and cloud," in *Proc. ACM MobiCom*, 2020.

[38] K. Hsu *et al.*, "Couper: DNN model slicing for visual analytics containers at the edge," in *Proc. 4th ACM/IEEE Symposium on Edge Computing (SEC)*, S. Chen *et al.*, Eds., 2019.

[39] L. Lockhart *et al.*, "Scission: Performance-driven and context-aware cloud-edge distribution of deep neural networks," in *Proc. 13th IEEE/ACM International Conference on Utility and Cloud Computing (UCC)*, 2020.

[40] P. Ren *et al.*, "Edge-assisted distributed DNN collaborative computing approach for mobile web augmented reality in 5g networks," *IEEE Netw.*, vol. 34, no. 2, pp. 254–261, 2020.

[41] The top 50 fastest computers in the VMW research group. [Online]. Available: http://web.eece.maine.edu/∼vweaver/group/machines.html

[42] C. Szegedy *et al.*, "Going deeper with convolutions," in *Proc. IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015.

[43] R. S. Sutton and A. G. Barto, *Reinforcement learning - an introduction*, 2nd ed., ser. Adaptive computation and machine learning. MIT Press, 2020.

[44] S. Beer, "The viable system model: Its provenance, development, methodology and pathology," *Journal of the operational research society*, vol. 35, no. 1, pp. 7–25, 1984.

[45] E. Barba, "Cognitive point of view in recursive design," *She Ji: The Journal of Design, Economics, and Innovation*, vol. 5, no. 2, pp. 147–162, 2019.

[46] S. Shlaer and S. J. Mellor, "Recursive design of an application-independent architecture," *IEEE Softw.*, vol. 14, no. 1, pp. 61–72, 1997.

[47] J. Touch *et al.*, "A recursive network architecture," ISI, Tech. Rep. ISI-TR-2006-626, 2006.

[48] J. D. Touch and V. K. Pingali, "The RNA metaprotocol," in *Proc. IEEE ICCCN*, 2008.

[49] J. Day, *Patterns in Network Architecture - A Return to Fundamentals*. Prentice Hall, 2008.

[50] J. Day *et al.*, "Networking is IPC: a guiding principle to a better internet," in *Proc. ACM CoNEXT Re-architecting the Internet Workshop (ReArch '08)*, 2008.

[51] J. D. Touch *et al.*, "A dynamic recursive unified internet design (DRUID)," *Comput. Networks*, vol. 55, no. 4, pp. 919–935, 2011.

[52] M. Salehie and L. Tahvildari, "Self-adaptive software: Landscape and research challenges," *ACM Trans. Auton. Adapt. Syst.*, vol. 4, no. 2, pp. 14:1–14:42, 2009.

[53] D. Weyns, "Software engineering of self-adaptive systems," in *Handbook of Software Engineering*, S. Cha *et al.*, Eds. Springer, 2019, pp. 399–443.

[54] S. Barati *et al.*, "Proteus: Language and runtime support for self-adaptive software development," *IEEE Softw.*, vol. 36, no. 2, pp. 73–82, 2019.

[55] S. Nastic *et al.*, "SDG-Pro: a programming framework for software-defined IoT cloud gateways," *J. Internet Serv. Appl.*, vol. 6, no. 1, pp. 21:1–21:17, 2015.