electronics

MDPI

Article

ASSIST-IoT: A Modular Implementation of a Reference Architecture for the Next Generation Internet of Things

Paweł Szmeja [1,*,†], Alejandro Fornés-Leal [2,†], Ignacio Lacalle [2,†], Carlos E. Palau [2,†], Maria Ganzha [1,†], Wiesław Pawłowski [3,†], Marcin Paprzycki [1,†] and Johan Schabbink [4,†]

1 Systems Research Institute, Polish Academy of Sciences, Newelska 6, 01-447 Warsaw, Poland
2 Communications Department, Universitat Politècnica de València, 46022 València, Spain
3 Faculty of Mathematics, Physics and Informatics, University of Gdańsk, 80-309 Gdańsk, Poland
4 NEWAYS Electronics, Science Park Eindhoven 5010, 5692 EA Eindhoven, The Netherlands
* Correspondence: pawel.szmeja@ibspan.waw.pl
† These authors contributed equally to this work.

Abstract: Next Generation Internet of Things (NGIoT) addresses the deployment of complex, novel IoT ecosystems. These ecosystems are related to different technologies and initiatives, such as 5G/6G, AI, cybersecurity, and data science. The interaction with these disciplines requires addressing complex challenges related with the implementation of flexible solutions that mix heterogeneous software and hardware, while providing high levels of customisability and manageability, creating the need for a blueprint reference architecture (RA) independent of particular existing vertical markets (e.g., energy, automotive, or smart cities). Different initiatives have partially dealt with the requirements of the architecture. However, the first complete, consolidated NGIoT RA, covering the hardware and software building blocks, and needed for the advent of NGIoT, has been designed in the ASSIST-IoT project. The ASSIST-IoT RA delivers a layered and modular design that divides the edge-cloud continuum into independent functions and cross-cutting capabilities. This contribution discusses practical aspects of implementation of the proposed architecture within the context of real-world applications. In particular, it is shown how use of cloud-native concepts (microservices and applications, containerisation, and orchestration) applied to the edge-cloud continuum IoT systems results in bringing the ASSIST-IoT concepts to reality. The description of how the design elements can be implemented in practice is presented in the context of an ecosystem, where independent software packages are deployed and run at the selected points in the hardware environment. Both implementation aspects and functionality of selected groups of virtual artefacts (micro-applications called *enablers*) are described, along with the hardware and software contexts in which they run.

Keywords: IoT; NGIoT; reference architecture; modular software; virtualisation; containerisation; orchestration; edge computing; distributed; enablers; data

Citation: Szmeja, P.; Fornés-Leal, A.; Lacalle, I.; Palau, C.E.; Ganzha, M.; Pawłowski, W.; Paprzycki, M.; Schabbink, J. ASSIST-IoT: A Modular Implementation of a Reference Architecture for Next Generation Internet of Things. *Electronics* **2023**, *12*, 854. https://doi.org/10.3390/electronics12040854

Academic Editor: Maysam Abbod

Received: 16 December 2022
Revised: 3 February 2023
Accepted: 6 February 2023
Published: 8 February 2023

1. Introduction

The Internet of Things (IoT) is a very broad area of research and practical deployments that, at this point in time, comes with as much history as novelty. In practice, the term can refer to any distributed system that includes hardware of some capacity and uses Internet technologies (even if only in internal communications, without necessarily being connected to the Internet). More refined definitions may acknowledge IoT only as a subdomain of cyber-physical systems. In popular perception, however, the Internet of Things must include a multitude of devices, which are relatively small in size and/or function that range from microchip implants and smart watches, to home amenities or heavy industrial machinery. This unprecedented scope opens the domain to innovation in multiple dimensions and inclusion of virtually any novel software or hardware invention.

Following the experiences gathered in projects, financed under its supervision, the European Commission introduced the term Next Generation Internet of Things

(https://www.ngiot.eu/, accessed on 1 December 2022) (NGIoT). Here, the traditional IoT is seen as (1) composed of decentralised systems (often including cloud(s)), (2) multi-paradigm (e.g., serverless, microservice-based, event-sourced, NoSQL etc.), (3) dynamic (with roaming devices, (4) changing network and resource conditions, evolving requirements), and (5) highly heterogeneous. NGIoT adds new concepts and technologies [1] that include, among others, 5G with software defined networking (SDN) and network function virtualisation (NFV), edge computing, broad use of AI (also on the edge), Tactile Internet with applications of virtual reality (VR) and augmented reality (AR), digital twins, data spaces (i.e., dedicated data management solutions), and potentially, distributed ledger technologies (DLT) for data privacy and security.

NGIoT had its genesis in the need to overcome the limitations of traditional IoT ecosystems, such as a lack of virtualisation, a lack of scalability, and the impossibility of easily incorporating new technologies and protocols (such as 5G or cloud-native principles). The selection of technologies and ideas included in the NGIoT vision was guided by observation of existing trends, focusing on those that would bring IoT into an all-inclusive space, and expediting any information technology effort, in any domain. Moreover, if successful, NGIoT is to become a universal entryway to the digital transformation [2].

One of the key problems in IoT is management of network traffic [3]. There are about 13 billion connected devices in 2022, and the amount of data they generate is expected to reach 73.1 ZB (zettabytes) by 2025 [4,5]. The risk of congestion in the network is not far away. Additionally, new kinds of NGIoT services and the implementation of the Tactile Internet [6] concept, requiring near-zero latency, will require the deployment of 5G/6G networks [7] with increased bandwidth capabilities in order to support new services such as AR/XR and indirectly will introduce additional traffic in the Internet.

When defining the key tenets of the NGIoT, it was decided that bringing this set of ideas into reality, given their high-level abstractions, requires a top-down approach. Consequently, works on the idea of NGIoT itself advanced in parallel with research and innovation actions. Their goal, among others, was to work on high-level architectures for NGIoT. As a requirement, such architectures have to fulfil not only formal requirements of robustness of documentation, but are also expected to be validated in real-world applications. In other words, the NGIoT architectures have to be implementable and solve practical, real-world problems.

The primary objective for the ASSIST-IoT (https://assist-iot.eu/, accessed on 1 December 2022) project, funded under the H2020 programme, is to propose an architecture that comprehensively realises the NGIoT concepts and guidelines. Importantly, in addition to the architecture, the project delivers the necessary, accompanying software (as a part of a reference implementation of the architecture), which was validated in four real-life pilot deployments spread across three domains: maritime port operation, construction worker safety, and smart vehicle monitoring. Moreover, the ASSIST-IoT implementation involves *both* software and hardware. The latter comes in the form of a highly modular and configurable custom edge node, which adheres to the general architectural principles, and on which parts of pilot deployments of the project are executed.

The core novelty of this work is the capacity to include NGIoT-inspired concepts and technologies into actual IoT ecosystems. The developed architecture is flexible, scalable, and modular—features supported by widespread use of virtualisation and reusable packages—and goes beyond a theoretical design. It puts a comprehensive tool set in the hands of the community, thereby avoiding so-called "domain silos" (i.e., inability to transfer solutions between domains). Till now, IoT RAs and their realisations were either too domain-specific (health, construction, home, etc.) or too complex to be easily re-used in multiple verticals. In contrast, ASSIST-IoT RA and tools are designed to make NGIoT compliant deployments accessible to all establishments, entities, and stakeholders. Through tools, documentation, and open APIs, ASSIST-IoT deployments can be kick-started, and (by virtues of flexibility and adaptability) later fine-tuned or adapted to better fit the dynamic conditions of a modern IoT ecosystems.

1.1. NGIoT Landscape—Related Work

IoT ecosystems materialise, among others, in the context of hardware manufacturing efficiency [8], processing power [9], energy consumption [10], and interoperability [11]. The unprecedented scope and dynamism of scope extends to IoT architectures. A recent (at time of writing, December, 2022) study [12] reported that despite an impressive body of work, "heterogeneity of devices, underlying technologies and lack of standardisation pose critical challenges in this domain". The same study elaborates on how varied the set of pertinent features is. Introduction of novel areas of research and technology, e.g., Tactile Internet [6,13], edge computing [14], and 5G networks [15,16], brings new problems. Technological challenges are overlaid over societal issues, such as flow of data and associated data privacy and security [17]. The newer the problem, the higher the chance that instead of standardised approaches, one finds a varied list of attempted solutions, each targeting the space from a different perspective [18]. Therefore, direct comparison of large, IoT projects is very difficult, as they differ in scope far more than they share common features.

The NGIoT initiative promotes research into new aspects related to the evolution of IoT and the development of a new concept of the IoT-edge-cloud continuum. Here, most recent projects publish results related to the realisation of this vision. Thus, the available body of work describing comparable approaches is relatively modest.

Researchers belonging to the project VEDL-IoT (https://vedliot.eu/, accessed on 1 December 2022) proposed a novel way of partitioning applications (not necessarily container-based) into serverless functions, with the aim of efficiently executing AI processes in innovative IoT devices (https://www.ngiot.eu/ngiot-report-a-roadmap-for-iot-in-europe/, accessed 1 December 2022). Thus, this work is not directly comparable with the discussed approach, as the advanced features related to orchestration in various hardware nodes are not considered.

The project TERMINET (https://terminet-h2020.eu/, accessed on1 December 2022) proposed an architecture to bring NGIoT traits to the IoT ecosystems, by leveraging, mostly, mobile-based concepts. Drawing from the open protocols of software defined networking (SDN) a virtualised variant of multi-access edge computing (MEC), vMEC, was devised to be run on base station-assimilable nodes. Here, applications such as AR/VR or Tactile Internet are supported. However, the complexity of the edge-cloud continuum and the orchestration of microservices workloads have not been comprehensively addressed [19].

The closest in scope architecture [20] was proposed by IoT-NGIN (https://iot-ngin.eu/, accessed on 1 December 2022), where Kubernetes is also leveraged to manage container workloads. A significant aspect of that research was the inclusion of unikernels workload management, connected to cloud-native storage by ceph and rook. However, specific mechanisms were not declared. Furthermore, a focus on federation of devices and communication among them was reported, and offloading was the emphasis of smart orchestration among heterogeneous nodes. Nevertheless, IoT-NGIN considers neither horizontal scalability of resources, nor federated learning.

An alternative approach to modern IoT is exemplified in the RAMOS [21] project. It follows the Meta-OS approach, which is a conceptual extension of an operating system to a large scope that covers many machines or components. Just like a traditional O,S it provides layers of abstraction over hardware and software, allowing for automated resource management, low-level device control, etc., albeit in a decentralised environment. This approach focuses on agents that represent physical orchestrable resources, rather than on management of virtualised workloads. However, at the time of writing, no implementation is available, the resources are claimed to be dynamically allocated to tasks. Note that this project, and the Meta-OS approach in general, do not address specific points brought about by NGIoT, e.g., Tactile Internet, 5G, and DLT (although it does not preclude their use).

ASSIST-IoT, in addition to being in-line with NGIoT, defined an architecture that is reaching beyond existing approaches. Specifically, it should be stressed that, at the time of writing, no all-encompassing solutions have been found, tackling at the same time smart orchestration of cloud, edge, far edge and IoT nodes, and including (natively) features

such as self-* capabilities, federated learning, or 5G compliance, in a single architecture. Thus, ASSIST-IoT delivers an extensible blueprint of a domain agnostic architecture for NGIoT, combined with a complete toolset, to actually realise IoT ecosystems. The three diverse pilot areas involved in the action, (i) port automation; (ii) automation, and (iii) construction and safety at work, demonstrate the flexibility and extensibility of the approach. The ASSIST-IoT architecture includes all the needed building blocks and will be the base of the IoT-edge-cloud continuum initiative that is being used as support for innovative AI/ML-enabled networking capabilities, new service deployment and support for IoT-edge-cloud continuum.

1.2. Structure of This Contribution

The rationale and the core aspects of the ASSIST-IoT architecture have been outlined in [22]. The aim of this work is to discuss how the proposed architecture is implemented in practice. In the text, key information about how the IoT ecosystem is to be divided into manageable (and controllable) parts is presented, along with technologies used for actual management of software and hardware components. Mechanisms of control over virtual, decentralised environments are presented, including network and resource management.

In this context, the remaining parts of this contribution are organised as follows. To start with, the ASSIST-IoT architecture is briefly introduced in Section 2. This is followed, in Section 3, with a more in-depth description of specific hardware and software components, developed following the principles outlined in the architecture. First, the general principles of software development are presented in Section 3.1, followed by description of virtualisation in Section 3.2 through Section 3.5. Next, the role of hardware in the architecture and specification of custom hardware nodes are discussed in Sections 3.6 and 3.7. Examples of already implemented software packages are reported in Section 3.8. The real-world verification in pilot use-cases is discussed in Section 4. Finally, main conclusions and lessons learned thus far are presented in Section 5.

2. Architecture

An architecture, understood as a description presented in diagrams and supported by the documentation, provides context and facilitates understanding for an IT system. A reference architecture (RA) lifts such the descriptions to the meta-level. The need for a well-defined architecture is most apparent in large-scale systems, with complex interconnections between elements, such as real-world IoT ecosystems. Consequently, a reference architecture can truly show its strengths when individual parts represent heterogeneous domains and solve complicated problems. In essence, a RA is used to instantiate specific system architectures, which guide implementation and deployment of software and hardware. Introducing a RA helps managing complexity of any large and/or growing system and avoids pitfalls when redesigning or modifying existing deployments. This section briefly introduces the core concepts behind reference architectures in general, and the ASSIST-IoT RA specifically. Readers interested in more details are invited to consult [22], technical reports (https://assist-iot.eu/publications/, accessed on 1 December 2022), and the project deliverables (https://assist-iot.eu/deliverables/, accessed on 1 December 2022), which discuss the RA in greater detail.

2.1. Reference Architecture

Building complex systems requires making an enormous number of vital design decisions. Here, a good RA should help with making them and guide the developers and designers through system implementation. In this context, the RA consists of two categories of documents. (1) *Descriptions* of IT systems, including structures, models, design language, and guidelines. (2) *Blueprints* to design, develop, deploy, use, and maintain said systems. Moreover, the RA should be technology and domain-independent, and abstract and flexible. However, it should also be practical and useful, which is known to be particularly difficult to achieve.

In this context, the ISO/IEC/IEEE 42010 standard (https://www.iso.org/standard/74393.html, accessed on 1 December 2022) provides a reference for description of software architectures [23,24], including those dedicated to IoT. It defines three core formal concepts that are the key to understanding RAs:

- Stakeholder: A person, team, or legal entity that has a concern (also referred to as "interest") in a system. Stakeholders can be technology (developers, administrators, maintainers, hardware manufacturers, network administrator or provider, security team, IT department member, etc.) and non-technology related (end-users, business clients, regulatory bodies, local governments, etc.).
- Concern: An aspect of interest to stakeholders that is related to the RA. It may include needs, goals, business requirements, quality attributes and assurances, dependencies, responsibilities, etc. (for instance: allowing interoperability with systems of various vendors, ensuring traceability and accountability of data origin, treatment, and trustworthiness). It is similar to the usual concept of "requirement" but may tackle a substantially wider scope (for instance: data-related concerns may include large number of requirements, related to functionalities expected from the architecture, originating from multiple stakeholders).
- View: A diagram that depicts a selected part of the architecture, to illustrate how a specific group of concerns is addressed. Views are necessary, because RAs cover "too much" to be captured in a single "representation". The core RA views are, typically, (i) logical (or Functional), (ii) data (or Information), (iii) development, and (iv) deployment and operational views [25].

2.2. IoT Architectures—Related Work

Historically, IoT-A [26] was the first project that proposed a reference architecture for IoT. It stipulated to organise modular components into functional groups, namely: IoT process management, service organisation, virtual entity, IoT communication, security, and management. Other RAs used similar approaches, but with different module/layer definitions, such as ITU-T Y.2060 [27] and WSO2 [28]. In time, RAs evolved to include more advanced features, e.g., for addressing edge computing and virtualisation, such as OpenFog [29], ECC RA 2.0 [30], LSP 3D [30], and AIOTI HLA [31]. Moreover, the latter separated functionalities from properties and/or cross-cutting concerns. Separately, there is a set of RAs that include concerns related to industrial processes, such as IIRA [32], RAMI 4.0 [33], and FAR-EDGE [34]. More detailed comparison of those, with the ASSIST-IoT RA, can be found in [22].

In general, it can be conjectured that there is no single architecture that suits all technical and non-technical needs of all potential NGIoT systems. There are simply too many topologies, access networks, protocols, devices, data types, and technologies involved. Still, a set of requirements, guidelines, and recommendations is necessary to lead the design, development, and implementation of NGIoT realisations, especially as the complexity of the requirements (and hence, the number of involved solutions) increases. As for that, there is a constant need to both evolve existing ideas and propose new and/or improved RAs. Since ASSIST-IoT RA aims to fill this gap, let us describe it in some detail.

2.3. ASSIST-IoT Architecture

Large numbers of RAs remain conceptual exercises that have never been fully implemented. On the other hand, complex systems are often deployed without an actual RA providing their foundation. These two observations point to the main novelty of ASSIST-IoT and the main contribution of this work. The uniqueness and advantage of ASSIST-IoT's solution lies in its completeness. It delivers not only a RA, but also its hardware and software implementation. Moreover, it has been validated in the real world, and can be adapted any heterogeneous use-case. In a single sentence, the ASSIST-IoT RA is not only focused on potential realisation, but also is founded on robust theoretical underpinnings.

Building on the experiences of existing reference architectures for IoT, the ASSIST-IoT RA adapts concepts from cloud architectures, to better serve the dynamic landscape of NGIoT. However, it should be stressed that one more important source of inspiration, of the proposed RA, is the requirements collected from the four ASSIST-IoT pilots (see [35] and Section 4 for more details). Hence, taking into account the breadth of scope and complexity of the pilots, it can be claimed that the proposed RA is also well grounded in real-world aspects of IoT ecosystems. Keeping this in mind, let us now introduce and briefly discuss the key design principles that shaped the ASSIST-IoT RA.

- Service-orientation. Implementation of features as independent (micro)services allows flexibility of use, enables a high degree of manageability, and provides support for many different technologies. Under this principle, the features can be technologically independent, developed separately, and deployed on a per-need basis.
- Containerisation. Using containerised software decouples it from the execution environment and allows for separate management and monitoring of individual services and the resources that they use to run. It also facilitates useful system-level features, such as replication, load-balancing, and failover mechanisms.
- Orchestration. Comprehensive management of services is provided by a system-level orchestrator that provides control over lifecycle, status, network interconnections of virtualised services. An orchestration service arranges the containers to best use their synergy, deploy services at the best possible place, and allows administrators to fully exploit the benefits of containerisation (i.e., system-level features).
- Cross-cutting functionalities. Providing a set of the available functionalities "globally" allows grouping common system-level tasks under a single distributed provider, which not only saves on development time, but also assures better control, and thus higher quality of the resulting solution. Moreover, it allows one to treat the system as "something more than a collection of services", which is needed to guarantee, e.g., system-wide security. Furthermore, it simplifies circumvention of common problem with microservices, where the same functionalities are implemented multiple times for the sake of service independence.

These foundations of design support the quality of the developed system, its manageability, and its production-readiness, among other things. Although the RA is independent of any specific technology and could be implemented using any software that supports its design principles, the ASSIST-IoT project also proposes a reference implementation. Again, the proposal was based on in-depth analysis of the state-of-the-art of implementation of complex IoT systems and analysis of ASSIST-IoT pilots. For instance, in the pilot deployments, it utilises Kubernetes (k8s) and Docker images, and follows standards defined by these technologies. While, obviously, the RA design documents are technology-independent, based on gathered experiences, this contribution provides more details of the concrete implementation on k8s. The choice of k8s was made to ensure quick deployment (and validation of the RA in the real world) and a level of support from the community gathered around this already mature but still expanding technology.

Architecture Overview

Let us now present more detailed discussion of the main aspects of the ASSIST-IoT RA, which has been illustrated in Figure 1.

As can be seen, the RA is divided into layers (called *planes*) and vertical (cross-cutting) blocks. The planes represent groups of separable and self-contained functionalities that are independent of other planes. The *verticals* constitute properties, or NGIoT features, that can have independent interpretations and realisations on different planes and can intersect planes, including elements from more than one. The planes are:

- *Device and edge*: physical elements that support an architecture realisation; from servers and edge nodes to sensors, IoT, and network devices (including gateways).

- *Smart Network and Control*: functions that facilitate network virtualisation and connectivity, such as MANO, and virtualised network functions (virtual firewall, SD-WAN, VPNs, link aggregation, etc.).
- *Data management*: functionalities related to data, from acquisition to sharing, fusion, aggregation, transformation, and storage.
- *Application and services*: functions to be consumed by end-users, administrators and/or external systems; they connect to functions from (lower) planes and to verticals, to offer applications for stakeholders.

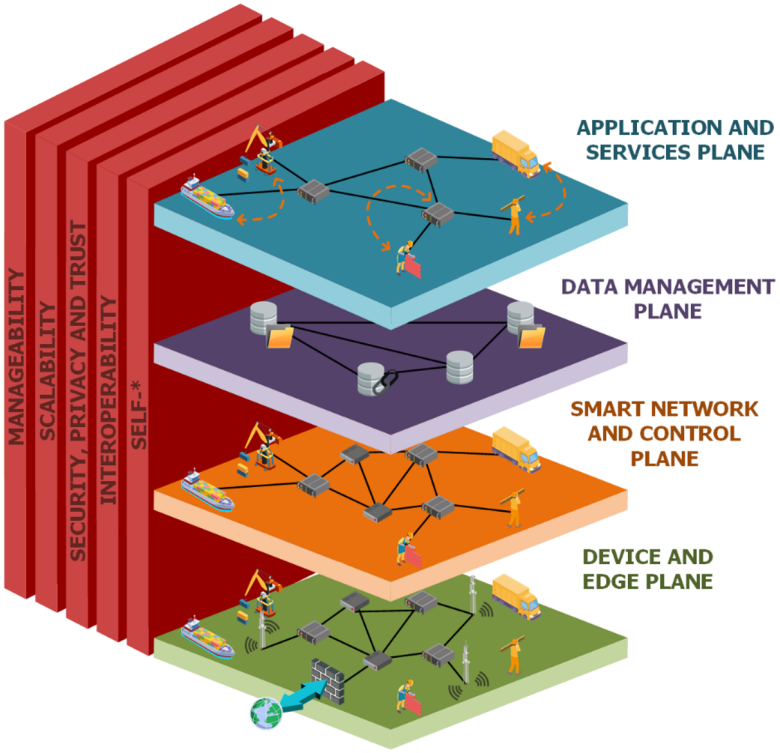

Figure 1. ASSIST-IoT layers and verticals—an overview.

It should be noted that the elements residing on individual planes should be treated metaphorically. Hence, for instance, we see a firewall protecting the ecosystem from the outside connections (on the *device and edge* plane). We also see that the same physical artefact (the ship) can interact with multiple planes. For instance, ship sensors are part of the *device and edge* plane; the ship is connected to the network on the *smart network and control* plane; it can also be part of an application, deployed on the *Application and Services* plane.

Next, the verticals are:

- *Self-**: properties related to its autonomy, or semi-autonomy, comprising operations that do not require human intervention (self-healing, self-configuration, self-awareness, self-organisation, etc.).
- *Interoperability*: properties that ensure that, (i) at the hardware level, equipment from different manufactures can communicate within a deployment, and (ii) at the software level, services can share data, thanks to common formats or protocols, or use of dedicated tools that facilitate needed aspect(s) of interoperability.
- *Security, privacy, and trust*: properties of the architecture, related to integrity and access restriction of data, and those guarding against malicious threats, among others.
- *Scalability*: properties ensuring proper system performance, and dedication of resources in case of change of operational, or business, conditions, or requirements. It comprises software, hardware and communication dimensions.

- *Manageability*: properties related to the control of the lifecycle functions of other planes and verticals, from their instantiation and configuration, to their termination. It also comprises management of devices and coordination of workflows.

The division of functions among planes and verticals aims at providing a high level of flexibility and manageability that fits the service-oriented approach and containerisation principles. It also delivers the right granularity of control, which is needed in the deployment of large complex systems. In practice, creating user-required application chains involves inclusion of elements from different planes and verticals, introducing a practical cross-dependency. Here, the difference between a realised architecture and a reference architecture becomes clear. In a deployed architecture, elements must depend on one another, to serve the end use-cases, despite their architectural independence in a RA. Nevertheless, modularity allows reuse of the same services in multiple deployments or multiple instances of the same service in a single deployment. The cost of management of individual services is offset by including tools from the *manageability* vertical (that support individual updates and management) and is small in comparison to the service-bus and to other, more monolithic, architectures. It also eases the understanding of the system by individual stakeholders, as knowledge about only few relevant services is likely to be enough for the user; complete understanding of the deployment will not be needed.

2.4. ASSIST-IoT RA Views

A reference architecture needs to consider multiple perspectives and include complete information about overlapping elements of the whole system. Including it all in a single set of models would be very difficult to understand for potential users [25]. Therefore, the presented RA is divided into so-called *views*, following ideas described in [36], where the 4+1 model was introduced. Here, the ASSIST-IoT architecture delivers the following views: functional, node, development, deployment, and data. Their brief summary follows. For a more detailed description, the readers are invited to consult the relevant project deliverable [37].

2.4.1. Functional View

The *functional view*, presented in Figure 2, defines independent functionalities grouped into separate planes.

It is a very general view that each plane defines functional "blocks", within which individual services are offered. Specifically, the functions offered on each plane are as follows:

- Device and edge: Contains physical elements (hardware), including user devices, edge nodes, and specialised hardware. For an edge-node to be considered ASSIST-IoT compliant, it must support the virtualisation host, on which the services are run. This plane is access-network agnostic, and specific hardware functions (i.e., hardware implementations/accelerators of AI algorithms) depend on particular use-cases. Here, note that the ASSIST-IoT project delivers specialised hardware for its pilot deployments (e.g., the (geo)localisation tag and the fall arrest device) and a general purpose edge node, as discussed in Section 3.7.
- Smart network and control: Groups network-related functions, including SDN and NFV. It abstracts the networking layer, so that services can communicate securely with each other and rely on this layer to take care of the networking for them. One of the most important functionalities that it offers is the orchestration (following the ETSI MANO [38] standard), which is described in more detail in Section 3.5. Moreover, this plane is focused on network (auto)configuration and provision of self-contained networks over VPN or SD-WAN, and connecting them with each other or the outside world.
- Data management: Groups' functions related to management, storage, governance, and transformation of data. It includes, among others, semantic functions, which annotate and transform data while taking into account their semantics. Mechanisms

for storing and transferring data in a dynamic environment with many potentially evolving data paths are also included.

- Application and services: Contains GUI applications and APIs meant to be exposed from the system and consumed either by external applications or by human users. Among others are dashboards and other monitoring tools, which report the state and performance of the system to the administrators and maintainers. The AR and VR interfaces are also included in this plane.

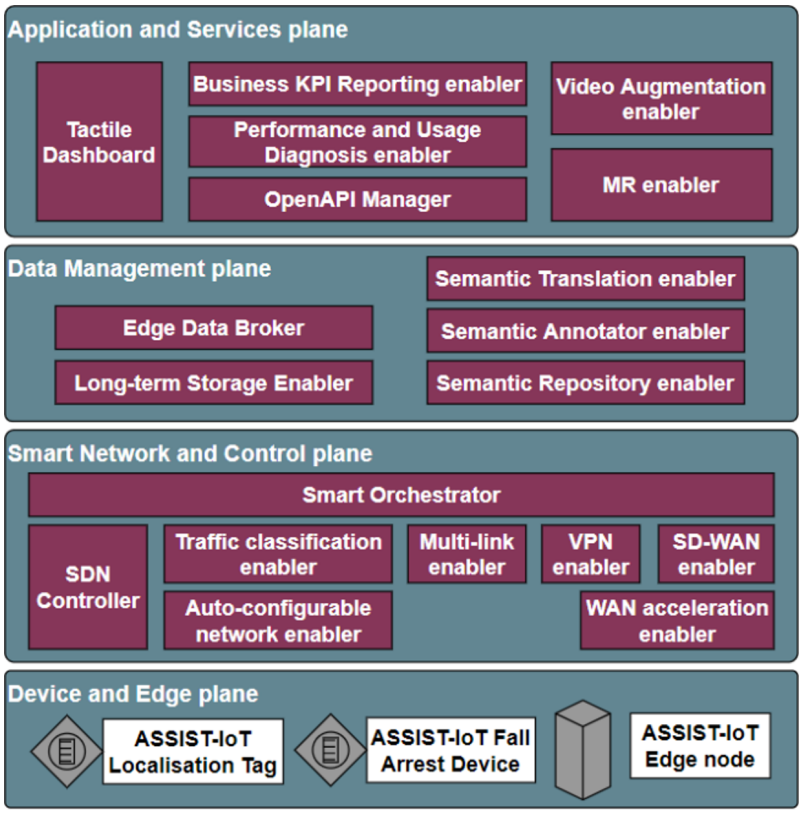

Figure 2. ASSIST-IoT functional view—planes.

Next, Figure 3 presents groups of functionalities considered in the verticals.

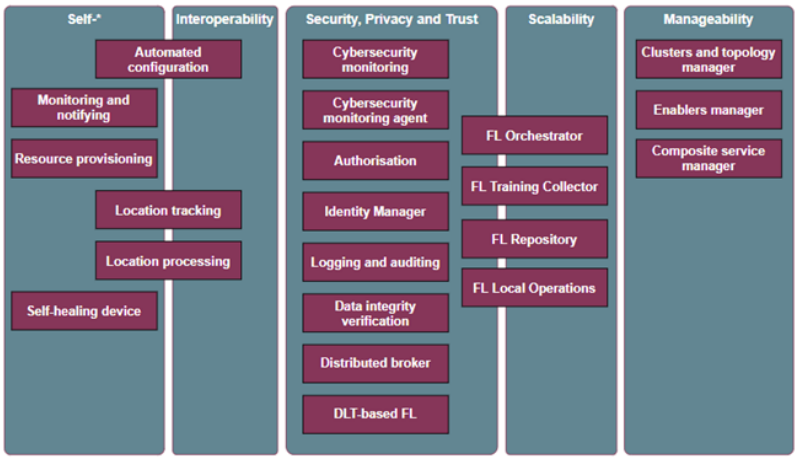

Figure 3. ASSIST-IoT functional view—verticals.

Here, note that because of their cross-cutting nature, such groups of functions cannot always be made independent and strictly separable. Therefore, the usefulness of the

functional view is more pronounced in description of the planes (Figure 2) rather than verticals (Figure 3), although both figures complement each other as parts of this view.

2.4.2. Node View

Moving closer to the hardware, the *node view* describes the nodes, which host the virtualisation environment in terms of both hardware and software (firmware). It is also concerned with the hardware-implemented functions, and the physical components, or modules, that realise them.

This view maps the logical elements, supporting the deployment of functionalities depicted in the functional view. It is not restricted solely to the hardware, as it includes also additional firmware and software elements. This view, usually, concerns both hardware developers and edge nodes or gateway providers. A more detailed description of a generic node of the node view is presented in Figure 4.

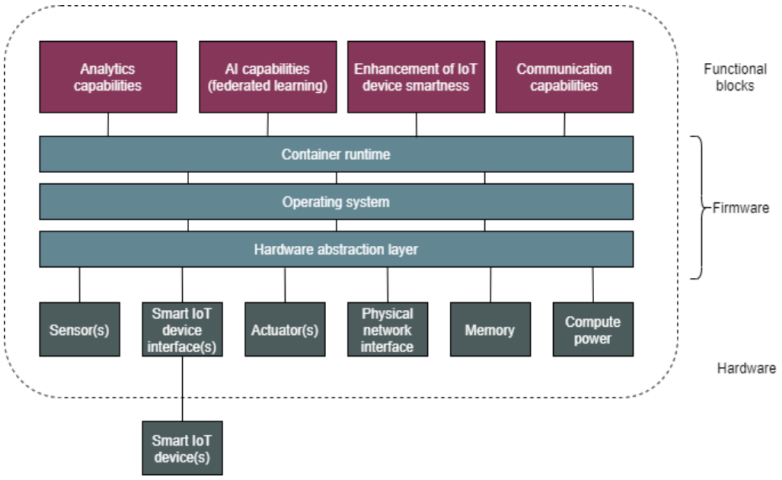

Figure 4. ASSIST-IoT node view.

2.4.3. Deployment View

The *deployment view*, outlined in Figure 5, presents the deployed functions in the context of physical nodes and their network interconnections.

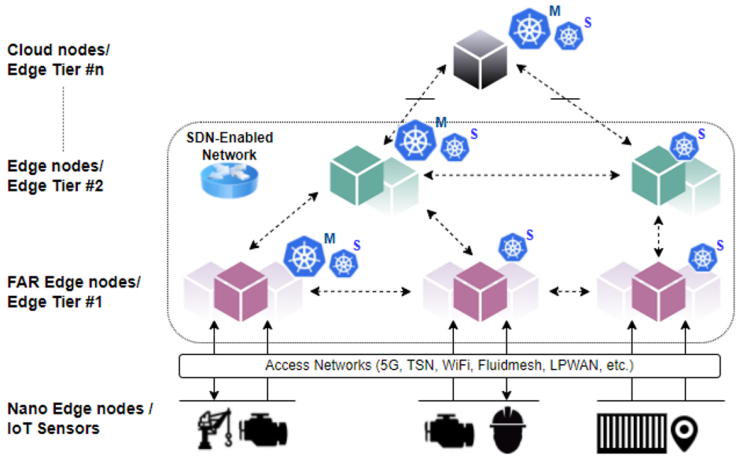

Figure 5. ASSIST-IoT deployment view.

Here, the topology of the network is represented, and critical points in communication between services can be identified. This view is divided into tiers, each representing a group of services and nodes, with specific distances between end-devices (user devices, sensors and actuators) and the cloud. Between the cloud tier and the sensor tier there can be any number of edge node tiers.

The placement of nodes, their interconnections, and the number of tiers are highly dependent on desired characteristics of individual deployments. Even though, by default, the deployment view presents all nodes as a single system, it must also support application chains that are independent and operate without reliance on each other. Note also that there are a number of services delivered system-wide and available to any service (e.g., management services). Thus, highly decentralised (and cloudless) deployments are also supported. Specifically, following recent trends, the edge-cloud continuum view assumes that data can be processed by any subset of connected nodes, and such processing does not need to involve cloud infrastructure(s). Moreover, physical nodes may host more than one service, and services may move between nodes (see Section 3.5).

2.4.4. Data View

The *data view* presents the interconnections between services with the physical layer abstracted away. It presents the communication as a high-level perspective on flow of data in separated scenarios. Instead of a single diagram, it proposes a visual design language in the form of "data pipelines" (an example is presented in Figure 6).

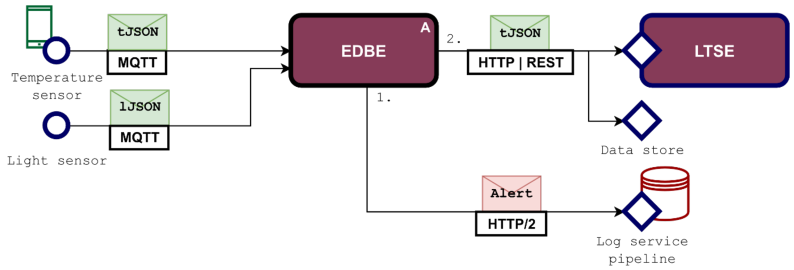

Figure 6. ASSIST-IoT data view—example data pipeline.

A data pipeline describes the lifecycle of data, often processed continuously, in a streaming fashion, from collection or emission, through transmission, transformation, to storage, or other forms of consumption. Here, data are conceptually divided into individual "messages" that travel through services and can cause a reaction or be transmitted further, possibly with changed contents or format. The data view captures data, with semantics relevant to scenarios and use-cases. This is not aimed at discussing technical aspects of the operation of the system, e.g., describing HTTP ACK messages, which are better captured on a lower level of abstraction, of UML flow diagrams.

2.4.5. Development View

Finally, to support developers directly, a development view is introduced (Figure 7).

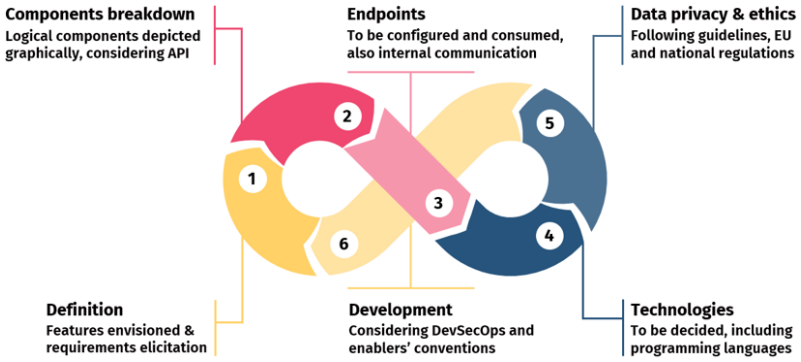

Figure 7. ASSIST-IoT development view—development process overview.

It is an adaptation of the DevSecOps methodology (Figure 8), which extends the DevOps approach, with a focus on the inclusion of security features in each iteration. It contains processes and guidelines that help software and hardware developers to follow

the principles of developing services under the RA. Recommendations for managing the complexity of developing a number of technology-independent services that may communicate through various interfaces are also included here. As understanding of the development process requires familiarity with specific software and hardware principles proposed by the RA (see Section 3); the details and guidelines are presented in Section 3.1.

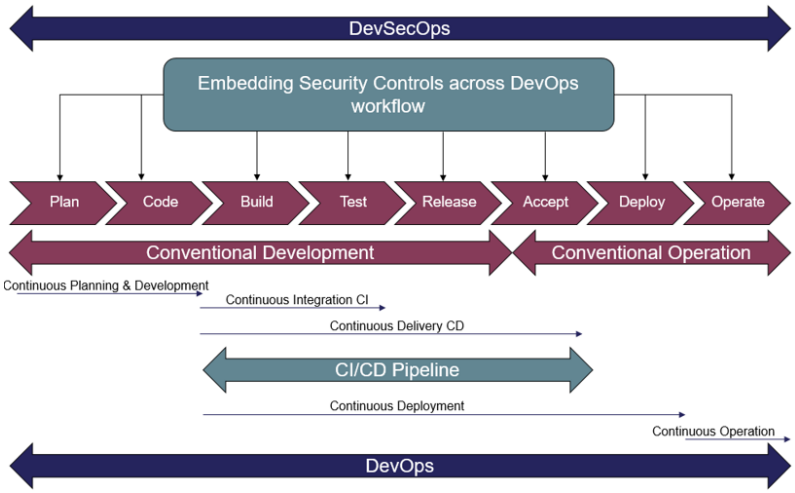

Figure 8. ASSIST-IoT DevSecOps process.

3. Enablers

Modularity, flexibility, independence, and distribution are the essential attributes of the ASSIST-IoT RA. Designing independent modules "on paper" is relatively easy, but one quickly encounters practical problems when realising the design. To address this, the RA introduces the concept of "encapsulated enabler" (or "enabler", for short) as a central part of its design. A detailed description of the concept and its role in the RA can be found in the ASSIST-IoT design documents [37]. Here we only summarise the key features.

In general, an enabler is the vessel through which modular services, discussed above, are actually delivered. It is a collection of software (and possibly accompanying hardware) components that work together to provide a specific functionality. Internally, an enabler can constitute of any set of components that communicate with each other while being isolated from the outside world. Externally, it is a "black box" that securely exposes selected services. In this sense, the concept is very similar to that of micro-services, with key differences, discussed in this and the following sections. Every enabler must adhere to the following principles:

- *Virtualisation*. Software components of each enabler are delivered as containers, along with automated deployment scripts. Therefore, enablers can be (re)deployed or removed as needed, and remain independent from one another. In practice, enablers may rely on each other to deliver functionality, e.g., system-wide security. However, they should be independently deployable and deliver core functions on their own. Enabler components are containerised, and should be considered as its internal parts with *no* direct external access and which are *inseparable* from the enabler itself. They are meant to offer "services" that are consumed *only* by other enabler components, unless explicitly exposed (see the "encapsulation" principle below).
- *Encapsulation*. Enablers can communicate with each other, and with other clients, *only* via explicitly exposed interfaces (e.g., REST API, and gRPC), jointly called "enabler interfaces". Enabler components cannot be interacted with directly from the outside of the enabler's internal scope. Informally, the encapsulation presents an enabler as a black box, the state of which cannot be directly influenced, and that can be communicated with only through specified channels. In essence, components of an enabler must be placed in a secure environment with a network perimeter (as in a

DMZ network). Being run in dedicated containers, components are separated from the host system. This supports the flexibility of internal component communication and implementation, without compromising standardisation, documentation, and security of consumer-facing communication channels (REST APIs, streams, etc.). Regardless of division of an enabler components between containers, an enabler should be orchestrable as a single package that preserves internal communication security, even if its components are deployed in different hardware or operating system contexts.

- *Manageability*. Enablers should expose metrics and logs. Even though the RA itself is open in this regard, ASSIST-IoT proposes to follow the de facto conventions. While cluster and container metrics (i.e., hardware and network resource usage) can usually be gathered by the container orchestration framework administrator, applications must expose their metrics individually. Following the proposed design, enabler metrics are exposed via a dedicated endpoint, using Prometheus-compatible format, whereas logs are sent via `stderr` and `stdout` interfaces. In addition, the general state of an application in a container is accessible also following pertinent standards, exposing endpoints that report health, i.e., the "liveness" (if the application still runs) and "readiness" (if the application is ready to serve or receive traffic) probes.

Following the described principles, enablers can respond to the needs and requirements originating from the ASSIST-IoT architecture. The blocks of the functional view (Figure 2) are implementable as enablers, bridging the gap between theory and practice. Modularity of the architecture is achieved by splitting functionalities into enablers. Hence, only the enabler relevant to a particular scenario needs to be deployed. Naturally, multiple enablers may be deployed and conceptually grouped together, into applications, synergistically offering more than individual enablers. The lack of a single central component, such as a cloud system, makes the conceptualised system highly flexible and able to quickly respond to changes that may materialise after the deployment. Deep containerisation supports joint use of multiple heterogeneous technologies that can communicate using standardised protocols. Note that such high level of virtualisation, and of containerisation, necessitates the deployment of a suitable virtualisation and orchestration environment. This aspect is discussed in the following sections.

Figure 9 presents a generic diagram of an enabler, with some internal components, external services offered through the enabler interface, and a secured internal scope, containing only the enabler components.

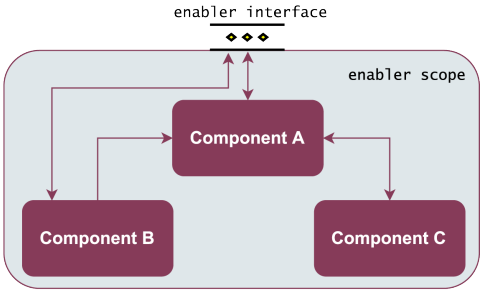

Figure 9. General enabler design overview.

It is important to note that enablers are not microservices. Instead, they should be conceptualised as "micro-applications". This name refers to recognisable similarities with microservices (as discussed above), and not to the perceived size or functionality limitation of an enabler. Just like microservices, enablers can be orchestrated and monitored while "hiding" their internals. Enablers are, however, not limited to a single function; they place no requirements on state management and can implement their components following any design pattern. For example, some/all enabler components may happen to be implemented as microservices. Here, microservices that should be available externally become a part of the enabler interface, while others are not exposed and are used only internally.

Internal enabler components' design and functionalities are left to the developer. An enabler may have an internal storage system, a server providing a frontend, a separate one for the backend, and even a load balancer. However, load balancing can also be achieved using the chosen container orchestration system. Here, it is crucial that the internal network of an enabler is accessible only via exposed APIs (realising the encapsulation principle).

3.1. Enabler Development

One of the outcomes of the ASSIST-IoT project is a set of enablers that deliver functionalities presented in Figures 2 and 3. Note that the description of the concept of an enabler was purposefully expanded in the design documents, to facilitate development of enablers outside of the project. This future-proofs the RA, allowing it to respond to evolving challenges and requirements.

In addition to the principles outlined in Section 3, an enabler design process, technically included in the development view (Section 2.4.5), has been proposed. The steps of this process (see Figure 7) are as follows:

1. The design of an enabler starts with its definition and elicitation of requirements. As in the case of the RA, the requirements should address the actual needs, expected features, and technological constraints, taking into account the business scenarios (however, they could also be generic, applicable to different verticals). Specifically, in this process, it is necessary to:
    - Depict the main objective of the enabler with a set of key, foreseen characteristics.
    - Describe user and business-related functional and non-functional requirements.
    - Analyse the general constraints of the (hardware and software) environment in which it is expected to run.
    - Outline use cases and scenarios in which it will apply. Knowing what stakeholders and end users expect from it can help consolidate its definition.
2. Once features are captured, a breakdown of the internal, logical components of the enabler should be conceptualised. Ideally, each component should be in charge of providing an internal feature of the enabler (even though, during implementation, some features might be grouped and executed by a single container).
    - It is highly recommended for the component structure of the enabler to be depicted graphically, considering an API as the main interface to be configured and consumed (additional interfaces, e.g., pub/sub, can be defined if needed).
    - Additionally, describing the lifecycle of an enabler, and addressing its use cases in terms of (high-level) interaction of the internal components, eases further design steps.
    - An initial mock-up of the communication endpoints and methods that will be exposed with the API should be outlined.
    - Apart from inherent configuration and functional endpoints, ASSIST-IoT strongly recommends the provision of a set of common endpoints that should be present in all enablers (see Section 3.3) and using standards, such as REST.
    - Additionally, the internal communication among components has to be addressed (e.g., via gRPC, REST API, etc.).
3. Programming languages and technological solutions should be selected. In some cases, individual internal features might be already provided by the existing software, and if they fit the ecosystem under development, can be adapted to the project's needs. If custom components are needed, they should be developed considering resource optimisation (e.g., hardware resources may be scarce) and decentralisation.
4. If data are involved, considerations declared in the ASSIST-IoT Ethics and Privacy Protection Manual [39] should be followed. In addition, particular clauses for specific sectors and/or specific national-level regulations might apply, which must be studied and applied.

5. Once development starts, the methodology described in ASSIST-IoT DevSecOps Methodology and Tools [40] should be followed (i.e., DevOps embedded with continuous security assurance [41]). Here, high-level recommendations related to virtualisation and shifting towards Cloud-Native approach are:
    - Containers should be based on well-known and dependable images. Hence, verified Docker images from Docker Hub, provided by consolidated vendors, are recommended.
    - For very early building and testing processes, the initial integration of internal components can be based on Docker Compose files before moving towards k8s.
    - ASSIST-IoT stipulates that enablers' packaging should follow provided packaging guidelines and use Helm charts (see Section 3.4).
    - A CI/CD pipeline for automating the operations should be set up. Unit testing, code and security testing procedures, and packaging tools and repositories, should be provisioned and incorporated into the DevSecOps framework (e.g., GitLab, GitHub); and adapted to the working environment and the expertise of the development, security, and operations teams.

If DevSecOps principles are followed [40], the final result will be secure by design, prepared for edge-oriented environments, and ready to be deployed with the Smart Orchestrator (see Section 3.5). Since DevSecOps is an iterative process, the above steps (1 to 5) should be iteratively applied and evaluated, using insights gained during the design, implementation, and testing phases.

3.2. Virtualisation and Containerisation

In computing, virtualisation stands for the creation of a version of something (computing hardware platforms, network resources, storage devices, etc.), abstracting the underlying hardware and/or software. Regarding applications and services, this allows the creation of computing environments that host them, with virtual machines (VMs) and containers being the typical realisations (unikernels and serverless are the emerging alternatives; however, thus far, they have not reached the same maturity level). On the one hand, a VM is an entire virtualised operating system (OS) that runs on top of a hypervisor, which abstracts the underlying hardware resources, or on top of another OS (with some performance penalty in the latter case). On the other hand, a container is a unit of software that packages code and its dependencies so an application can reliably run across computing environments, utilising an engine that abstracts the (software) resources provided by the underlying operating system.

It should be noted that the current trend in computing is shifting towards cloud-native ecosystems, where microservices and containers play a key role. Commercial cloud providers offer multiple virtualisation models, including Functions-as-a-Service (FaaS), and users decide whether to deploy their application as interconnected functions, containers, or VMs, or can even be granted total control over "their machines". Of course, the latter two approaches have greater economic costs, as more resources are required. However, as the ASSIST-IoT architecture targets the complete computing continuum, from the edge (including devices) to the cloud(s), containers offer more benefits over VMs, as they consume less resources. Hence, they can be deployed on (more) resource-constrained nodes. Apart from being lighter, containers are more flexible and can be deployed much faster, while also being mature in terms of technology and community support.

Therefore, the decision of designing enablers as a set of containers is relatively obvious. Still, to be ready for production, enablers should be complemented with additional features, such as automatic healing and resource (up/down)-scaling operations. Container orchestration frameworks provide such features, facilitating adoption of the architecture and the implemented enablers. Here, solutions such as Kubernetes, Docker Swarm, or Apache Mesos can be in charge of (i) managing the deployment of containers, (ii) verifying their correct operation, and (iii) handling errors, which are particularly important when large numbers of containers and clusters are to be managed. Although the ASSIST-IoT architec-

ture does not mandate any specific container framework, K8s is strongly recommended for orchestrating the enablers, as it is the de facto standard. Moreover, it is more mature, and provides more features than its alternatives. For instance, Kubernetes (1) automates rollout and rollbacks; (2) if needed, it automatically scales services up or down, horizontally (i.e., more/less replicas) and/or vertically (i.e., more/less computing resources on the existing replicas); and (3) it performs periodic health checks against deployed services, to ensure they are running as expected. It should be stressed that while, technically, the ASSIST-IoT architecture itself could be instantiated on any container orchestration platform (GKE, EKS, Docker Swarm, etc.), the existing reference implementations use Kubernetes. Moreover, Docker Compose files are available for selected enablers, to run them independently of the rest of the architecture, without the encapsulation and other features not available in plain Docker or Docker Compose.

3.3. Enabler Endpoints

The ASSIST-IoT architecture recommends a set of conventions and best practices that enabler implementations should follow. They should also be applied for integration with essential enablers. For instance, use of both a monitoring and a logging stack is encouraged, for gathering relevant functional, performance, and usage data of the deployed services, which can be later used for debugging, error alerting and handling, resource optimisation, and further improvement of the overall performance of a given enabler. Specifically, the following endpoints are recommended for any ASSIST-IoT-compliant enabler:

- /health (GET method): This endpoint is responsible for collecting the status of the internal components of an enabler, reducing the need to interact with the k8s API. The implementation largely depends on the nature of the components and the mechanisms exposed by them to be interacted with, so a specific structure is not mandated. Status codes are encouraged, specifying the unready, or failed, component.
- /api-export (GET method): This endpoint includes two methods to be used as documentation (including examples) and as an integration mechanism with the Open API management enabler, which are considered essential (i.e., present in almost any deployment). The two methods are:
  - /openapi (GET method): Returns the Open API specification in JSON format, so users are able to download and consult it.
  - /docs (GET method): Returns Open API documentation provided by Swagger UI, letting users visualise and interact with the API's resources.
- /metrics (GET method): This endpoint provides relevant metrics representing enablers' current statuses to be collected by the monitoring stack. Considering cloud-native de facto standards, this endpoint should be developed following the conventions mandated by the Prometheus technology.
- /version (GET method): This endpoint returns the current version of the enabler, and should follow the Semantic Versioning Specifications (SemVer).

3.4. Helm Chart Generator

An enabler is likely to consist of more than one component. Each of them requires a set of manifests to work on k8s clusters, which implies that for a single enabler, several manifests might be needed. Thus, for practical reasons, enablers should be packaged following a common strategy, before being shared and deployed in the pool of managed clusters. There exist different technologies that could be used for this purpose, e.g., Helm charts, Kustomize templates, and Juju bundles. For the ASSIST-IoT reference implementation, Helm has been selected, as (i) it is the de facto standard, so there are many solutions with production-ready, maintained charts; (ii) it provides templating capabilities, so multiple values can be configured automatically, or by its users, for tailoring environment variables to the targeted scenario; (iii) it implements strategies for upgrades/rollbacks, so services are kept online during these updates, etc. The packaging process is referred to the prepa-

ration of the k8s resource manifests (chart templates), wrapping them as a compressed file, and uploading it to a chart repository. In essence, enclosing every enabler in a virtual, orchestrable package allows for high-level management of redundancy, replication, hot-swapping, high-availability, and scaling. Hence, features very important in cloud computing can be brought into the IoT-edge paradigm. Here, different strategies that could be followed for packaging the enablers can be seen in Figure 10.

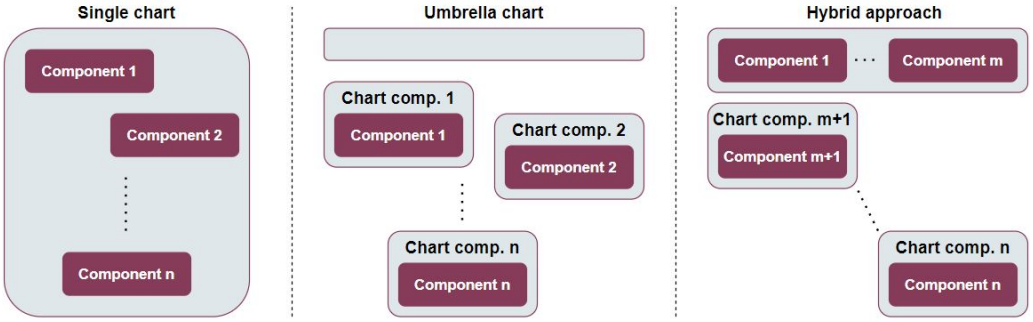

Figure 10. Helm chart structuring strategies.

First, a single chart could host the required manifests of all its components; second, an umbrella chart could point to the individual charts of each component as *dependencies*; and last, a hybrid approach could be applied, where the umbrella chart would hold manifests of some components while others are kept separately. Here, the third option (hybrid approach) is encouraged, so that third-party charts can be maintained separately and the manifests of custom developments are hosted in the umbrella chart. The structure of the hybrid approach consists of the files and folders presented in Figure 11.

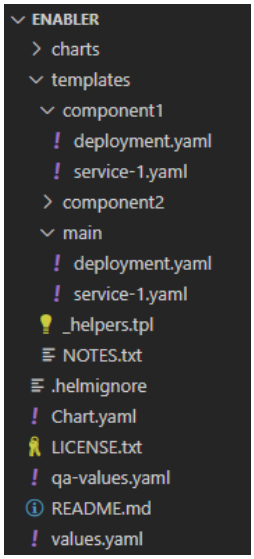

Figure 11. Proposed enablers' chart structure.

It is quite similar to the structure proposed by Helm but includes a separate sub-folder per a custom component in the *templates* folder, keeping the third-party charts in the *charts* folder. The default values of both custom-made components and dependencies are included in the *values* manifest. All the components must include at least two templates, one for the workload type and another for the service, but they can include also additional ones (ConfigMap, Secret, PersistentVolume, PersistentVolumeClaim, Ingress, etc.), depending on the specific needs of the deployed system.

Aiming at easing packaging of custom developments, a Helm-chart generator has been developed. It works as a wizard, asking questions such as: (i) the name of the enabler,

(ii) a description of the enabler, (iii) chart and application version, (iv) the number of components, and (v) the numbers of jobs, cronjobs, and dependencies. For each of the components, it asks for its name, type (e.g., Deployment, StatefulSet, or DaemonSet), and the number of services (typically one, as only one port is used). For each dependency, the name, version, and repository is required, so they can be properly retrieved. Next, the generator provides a baseline chart structure with the number of components selected, a set of predefined manifests (workload and service) with the fields suitably populated, and the labels required to enrich the performance of the orchestrator. Typically, the user will take care of just a few fields of the *values* manifest (e.g., port of the service, specifying whether a component is an exposed interface or not) and the environmental variables particular for each component.

3.5. Enabler Orchestration

Enablers from different *planes* and *verticals* deliver different features of an NGIoT system. They are designed as sets of components, developed as containers, and deployed in a container orchestration framework installed on the computing nodes. Still, deploying them in a pool of virtualised, distributed clusters requires a lot of knowledge about the container platform and its options, plugins, and packaging formats. Hence, supporting tools are needed, especially when the numbers of clusters and services increase.

ASSIST-IoT proposes to use the Smart Orchestrator enabler for easing this process, focusing on automation and usability. Considered essential and most important for an architecture realisation, it has been implemented in line with the (core aspects of the) ETSI MANO specifications [38] and current trends towards virtualisation and cloud-native principles [42]. For this, the following aspects had to be taken into account:

- A central environment, in which all the pool of managed clusters (Kubernetes distributions) and enablers repositories are registered.
- Automatic provision of ETSI MANO compliant manifests (NS, VNF descriptors), extracting the required data from the enablers' chart manifests, without the need for human intervention.
- A set of components to deploy other enablers, either manually or based on the the scheduling policy selected by a user.
- Logic for keeping, or deleting, enabler-related stored data, when this is to be deleted.
- Automatic application of network-related rules to allow, or to prevent, communication between enabler components, following the encapsulation principles (see Section 3).
- A set of components to manage large volume of clusters, where mobility is also a key factor (i.e., IP addresses of the clusters are expected to change).

The orchestrator, components of which are deployed in the top hierarchy of the managed computing continuum, exposes an API to be consumed. Aiming at easing its usage, it is installed along with a set of manageability enablers, which allows making use of a graphical, user-friendly interface. It should be stressed that the managed Kubernetes clusters should incorporate a set of plugins so the orchestrator can offer its features, such as Helm (as a package manager), Cilium (for controlling communication aspects at k8s network layers 3, 4 and 7), CoreDNS (for managing the DNS addresses of the services), OpenEBS (for managing storage classes), and Prometheus (for having the metrics required by the scheduler to make decisions). The components of the Smart Orchestrator can be seen in Figure 12.

In the discussed approach, the MANO framework is extended with: (i) a high-level API, witch abstracts and extends the features of the MANO API; (ii) a scheduler, which selects the optimal virtualised infrastructure manager (VIM) of the computing continuum, based on the policy chosen by the user; (iii) a network component, which acts over the networking plugins of the VIMs for allowing/denying specific communication; and (iv) a monitoring server, which jointly with the distributed agents, collects data from VIMs and enablers, to be consumed either by the user or by the scheduler's intelligent frameworks.

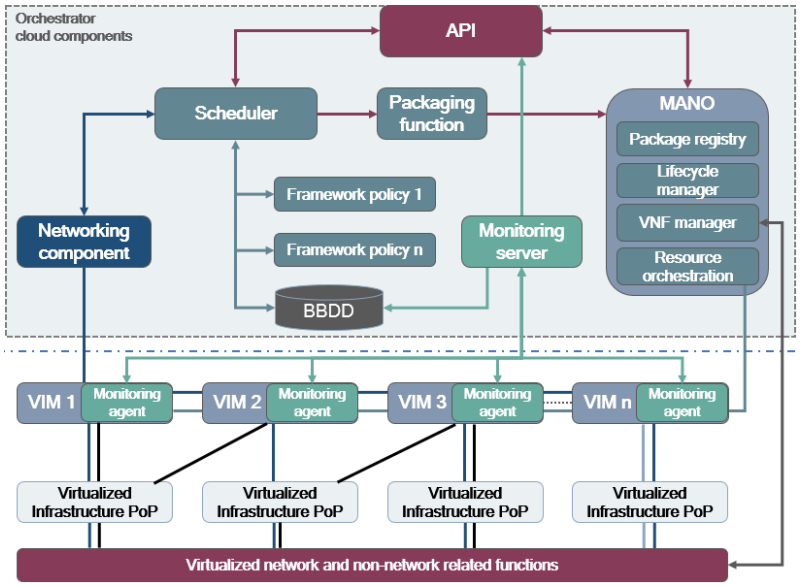

Figure 12. High-level smart orchestrator architecture.

Encapsulation

Using a combination of network plugins (Cilium) and packaging (Helm), the Smart Orchestrator can automatically configure the network environment to enable or disable connections between individual k8s pods. If an interface should be exposed "to the world", the Helm chart should explicitly set the value of the "exposed" flag to "true". Otherwise, no external connections will be allowed. On the other hand, recall that components of a single enabler can communicate by default. Here, restrictions should be left to the developers. Naturally, an administrator, having full control over the network and k8s management, can introduce custom modifications. Hence, the automatic setup is in line with the encapsulation principle, where an internal enabler communication is isolated and only selected interfaces are exposed.

With the Smart Orchestrator, the encapsulation is achieved without extra development work. Note that although this could be implemented on any alternative technology instead of k8s, the complexity of implementation and the practical consequences of such a decision would vary. In this context, note that choice of k8s ensures that key security features that are necessary for encapsulation are available and maintained by the community.

3.6. Embedded System Setup

Before enablers can be deployed and orchestrated, the virtualisation environment needs to be prepared with hardware to run on. Given the high level of virtualisation, hardware across the edge-cloud continuum can be used to host the containers (and thus, enablers). This includes cloud systems and continuum nodes with varying degrees of computational power. An overview of how containers and virtualisation hosts are placed on the hardware is presented in Figure 13.

While the cloud systems offer a lot of power, at the cost of physical distance to the clients, edge nodes require a separate approach to support differing use-cases that may require specific hardware components, interfaces, or computational capabilities. In this context, ASSIST-IoT delivers its own edge node called the GWEN (Gateway Edge Node). With the goal of flexibility and adaptability in mind, GWEN is a modular board with interchangeable components that communicate through standardised interfaces with the motherboard. The following sections discuss particular hardware questions and issues related to the IoT edge nodes. The GWEN itself is described in more detail in Section 3.7.

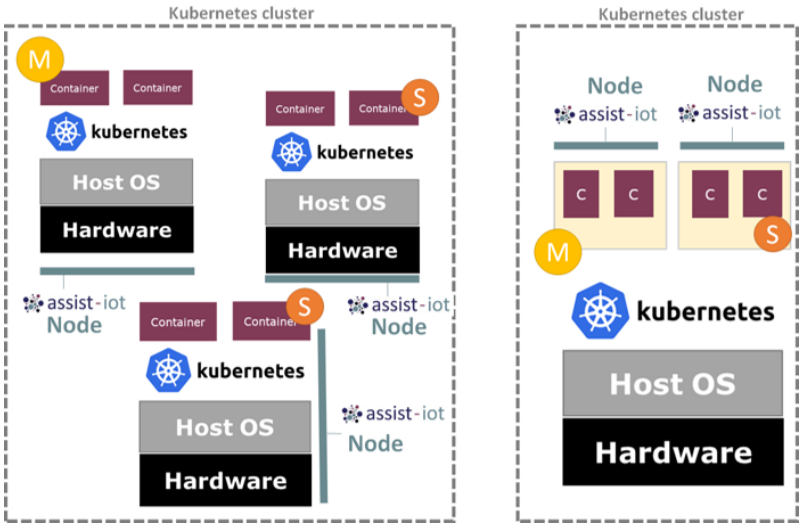

Figure 13. ASSIST-IoT hardware-software relationship overview.

CPU Architectures

This section considers two issues related deployment of workloads in an NGIoT system: (i) hosts with different CPU processing architectures and (ii) use of hardware acceleration for workloads requiring intensive computation. The majority of operating servers on the market, in addition to operating on Linux OS, contain CPUs with x86 or x64 sets of instructions (such as Intel and AMD architectures). However, especially in constrained devices (e.g., RaspberryPis and pico-clusters), other processors, such as ARM, which deliver better power-efficiency, are also used. Moreover, their popularity is systematically increasing.

Usually, services (i.e., enablers, or specifically, Docker images specified within the Helm charts) have been developed and built for specific processor architectures. Hence, if one has been built for an x64 architecture, it may not function properly on an ARM processor.

As mentioned, the pragmatic truth is that almost every piece of software which is becoming an enabler is designed and built making use of baseline Docker images. Sometimes, these must be very raw (e.g., Python, alpine) or even custom, but on most occasions, the enablers are rooted to inherited images that were built for a specific processor architecture. It may be that an enabler should be capable of running on nodes equipped with different processors (e.g., x64 vs. ARM). To cope with this variability, three lines of action are envisioned:

1. When enablers will be deployed *only* either on powerful servers or in resource-constrained devices, containers could be built only for the target instruction sets.
2. When enablers could be hosted in different spots within the continuum, their images should be built considering, at least, the most common CPU instruction sets (e.g., amd64, arm64v8, and arm32v7). If feasible, multi-architecture images (a feature well supported by Docker) should be delivered. The feasibility of this approach depends on the availability of software libraries compiled for more than one architecture. If all used dependencies are compiled for multiple architectures, then delivering a multi-architecture images requires low effort and is recommended.
3. Since many enablers will "live" within resource-constrained edge nodes (for instance, extensive historic data storage would not fall under this case, but a data dispatcher would), the edge equipment for ASSIST-IoT deployment (the GWEN) has been delivered with an ARM processor on board (see Section 3.7).

The second approach (above) is the most popular in open source solutions available in the market and is suggested for realising the enablers' images. In selected cases, such as the Smart Orchestrator and the long term storage enabler, that are not recommended to be run on (edge) nodes with constrained resources (because of their importance for the

deployment), compilation for a single target is acceptable. Otherwise, it is discouraged to target only a single instruction set or hardware. However, it may be unavoidable if very specific single-architecture-only libraries are needed. One should also take into account specific hardware requirements, such as cUDA cores, that may outweigh all other considerations and influence the enabler package targets.

When utilising Docker as the underlying container runtime, container images can be built for multi-architectures. Hence, the needed ones are compiled into the same image. To that end, Docker provides a tool named *buildx*. A drawback of this approach is the increased compilation time, but the trade-off clearly favours incorporating multi-architecture automatically using the CI/CD pipeline's process. Obviously, when images run on a given device, the container's runtime will select the right architecture.

Moving to acceleration capabilities (GPUs, FPGA's, etc.), containers cannot access this kind of hardware without extra installations, and configurations performed within the host space. In the case of GPUs, providing access to the containers requires that the drivers are provisioned work on the host (meaning that it could run GPU operations within the operating system user space). Moreover, pertinent libraries needs to be made available, so that the container's runtime would be aware of this resource and make it accessible to the deployed workloads. For instance, specific steps need to be followed at the host operating system level so that k8s can leverage NVIDIA GPUs. Similar actions are also required when considering acceleration via FPGAs. These actions are usually specific to the acceleration hardware brand. Hence, there is no unified way to prepare the hosts. Therefore, it is not possible (at this point) to include generic embedding of such acceleration capabilities to IoT nodes so that enablers could rely on their availability.

3.7. Custom Hardware Nodes

To address the requirements of the *device and edge* plane of the architecture and above considerations concerning use of enablers, a custom hardware node was developed. Here, the global goal was to deliver a light, edge-oriented computing (hardware) node that would be natively prepared to host services of NGIoT deployments (materialised in ASSIST-IoT enablers). The determination and choice of components resulted from analysis of common IoT use-cases, with special attention to the ASSIST-IoT pilot deployments. Moreover, the need to support the virtualisation host, as part of a virtual cluster, was also considered. The resulting hardware node is called GWEN, which stands for Gateway Edge Node.

The hardware of GWEN follows the principles outlined in Section 2.4.2 and Figure 4. In general, it consists of computer power, memory, physical network interface, smart IoT device interfaces, and dedicated sensors and actuators (if needed). It can act as a hub (through its physical interfaces), to which cooperating IoT components, sensors, actuators, networks, etc., are connected. Their functionality is then enhanced with the onboard computer power, memory, and storage—and made available to the virtualisation host. This part, in turn, runs the containers that provide "northbound" intelligent functions to the rest of the system and act as services exposed from the *device and edge* plane. Figure 14 presents a general block schematic diagram of GWEN, and Figures 15 and 16 depict the carrier board from top and bottom, respectively.

Selected interfaces are implemented on the carrier board and are an integral part of GWEN. Other interfaces are implemented as expansion modules and can be swapped as needed. What follows represents a default configuration of the hardware.

It should be stressed that GWEN was designed with modularity in mind. Given the range of possible IoT use-cases, there is no single catch-all hardware solution capturing all possible interfaces. To address this issue, interchangeable GWEN modules are used. For instance, even the CPU can be swapped. The current selection of interfaces and components, placed within the GWEN, answers requirements of the ASSIST-IoT pilot use-cases, and represents only a demonstrative example of components that GWEN supports.

Some functionality of GWEN is implemented through proprietary add-on expansion modules, mounted at the bottom of the carrier board (Figure 16). Size, a board-to-board

(B2B) connector, connector pinning, and mounting hole positions are fixed and positioned in a way that two sizes of modules can be inserted. An example configuration of mounted add-on expansion modules is visible in Figure 16.

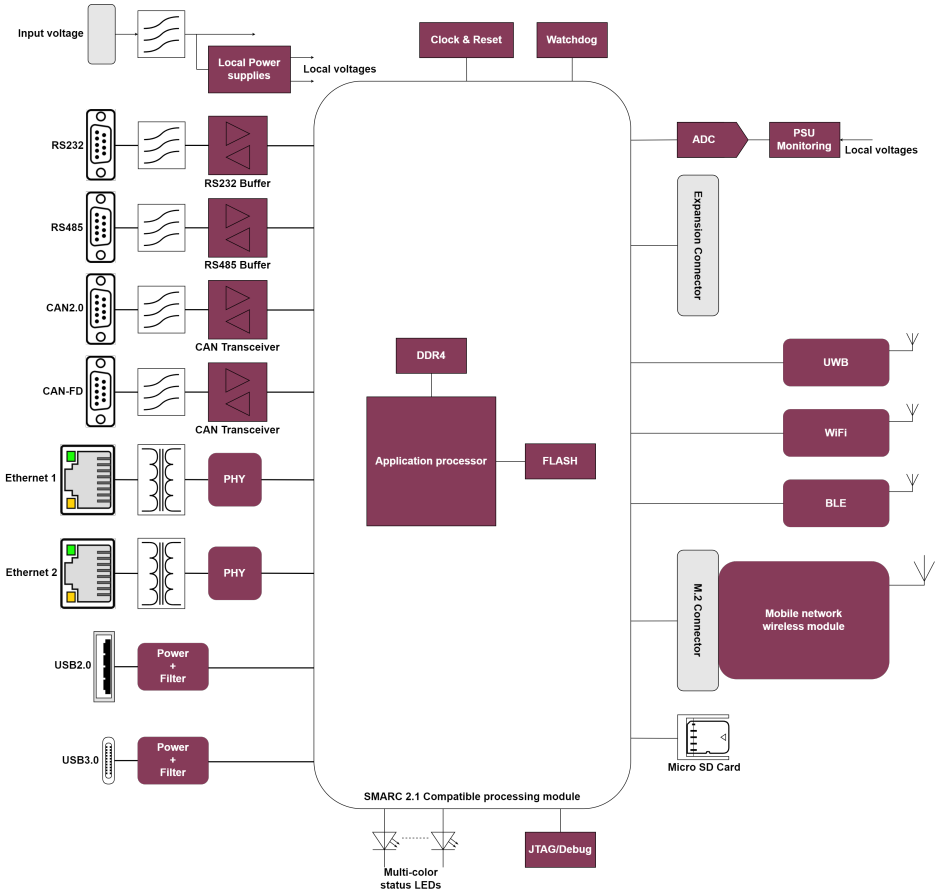

Figure 14. GWEN components overview.

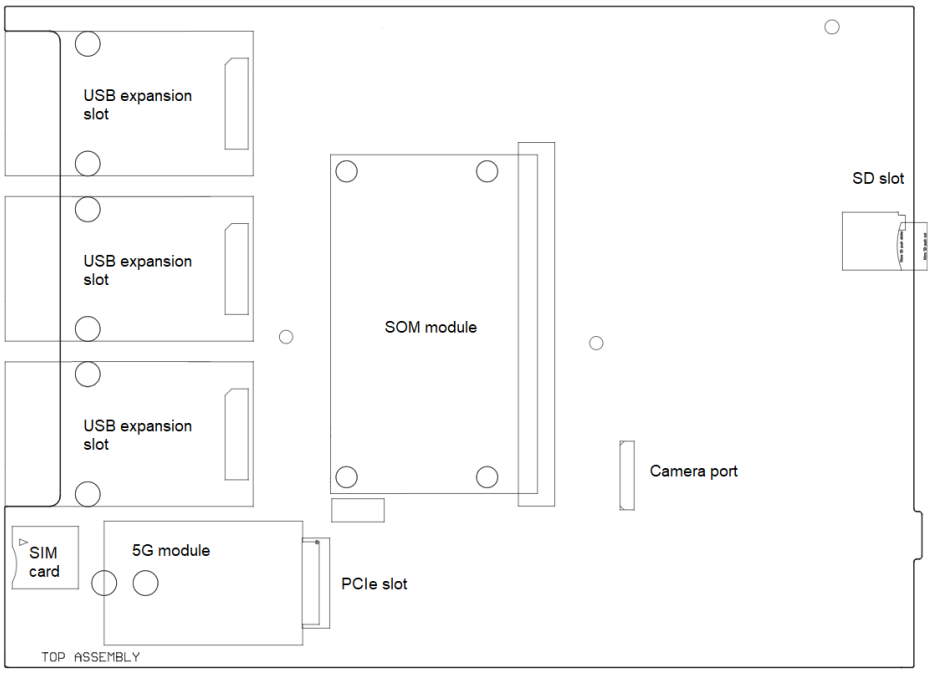

Figure 15. GWEN carrier board—top view.

The input voltage range for GWEN was initially set to 12 V ±5%. However, to support the project pilot, where GWEN was used in a car and connected to its onboard power supply, this voltage tolerance range was raised to ±20%.

With regard to the CPU architecture selection, GWEN went through a comprehensive analysis. For instance, the CPU was based on ARM, as enablers designed for running in the edge will be compiled for this architecture. In particular, it was decided to rely on a system on module board (SOM) with a standardised SMARC interface, so that specific configurations of the processor could be adjusted, if needed. Therefore, the chosen model of ARM processor for the current version of GWEN was the i.MX 8M Quad CPU @1.5 GHz. The SOM module allows different memory configurations, and addition of NPU or FPGA, in combination with the ARM processor. On the GWEN, multiple IOs and expansion slots are foreseen to allow a modular setup of the system, allowing CAN, 5G, or WiFi to be deployed. This aligns the design with the trends in NGIoT and supports modularity and scalability of the ASSIST-IoT RA. All I/Os are connectable over USB and SPI interfaces with the processor, whose kernel was updated from the original commercially available one.

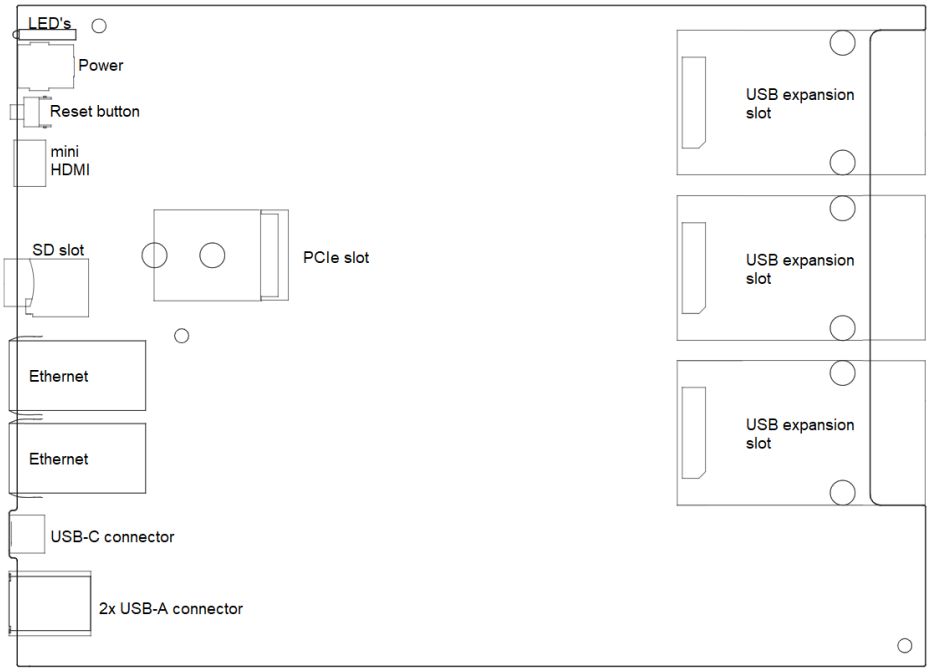

Figure 16. GWEN carrier board—bottom view (flipped horizontally).

The actual hardware (Figure 17) delivered by NEWAYS (partner of ASSIST-IoT project). The processor has an updated 5.4.7. NXP imX kernel, based on Linux Yocto 5.15, supporting Qualcom X55 chipset and a Quectel 5G module. Here, the bootloader was also modified to support booting from an external USB drive.

To provide more details of the hardware of the GWEN, the specifications of the different parts of the electronic components are given, as follows.

3.7.1. Wired Interfaces

For the wired interfaces, a generalised approach was chosen. Six dedicated interfaces, based on USB 3.0, were implemented for dedicated module boards, such as a CAN interface board. There were also two M.2 interfaces, based on PCIe for modules, such as 5G, WiFi/BLE, or SSD. In Figure 17, on the top left, a 5G (sub-6 GHz) module has been inserted, and on the top right, there are two CAN automotive modules.

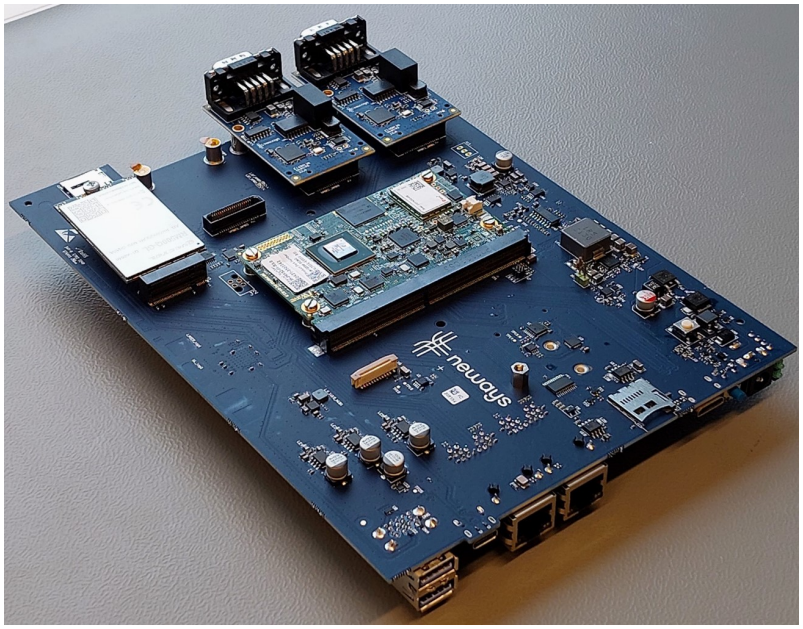

Figure 17. GWEN without enclosure (November 2022 version).

3.7.2. Wireless Interfaces

Based on the wired connections, wireless connections can be built, such as communication and/or localisation interfaces, which were used in the pilots of the ASSIST-IoT project. The *UWB* (ultra wide band) interface was built upon the USB wired interface and allows localisation. It can also be used for bi-directional communication, e.g., communication with the tags. This enables GWEN to act as an anchor. Key specifications of UWB are: (1) fully interoperable with IEEE 802.15.4 HRP UWB; (2) ranging technique based on IEEE802.15.14z, and (3) position accuracy <50 cm. Other Wireless interfaces, such as WiFi, 5G, and Bluetooth low energy, can all be connected via USB 3.0 or M.2 PCIe interfaces, allowing high speed connections between the communication interface and the CPU.

3.7.3. Computing Power

A processing module with a SMARC (Smart Mobility ARChitecture) interface was used. SMARC 2 is a standardised small form factor computer module definition. Since this is a standardised interface, any SOM (system on module) with a SMARC interface could be connected, allowing computing power and memory to be adjusted as needed. For instance, for one of the pilots, an artificial intelligence (AI) component was required. Therefore, an alternative SOM module which includes an neural processing unit was selected.

3.7.4. Additional Support Functions

Aside from the electronics, the following supportive functions are provided:

- *Clock & Reset* generates clock signals required by different elements of GWEN.
- *Hardware Watchdog* receives periodic, small, priority signals from the firmware. If a signal is not received within configured timeframe, the watchdog initiates a hardware reboot. The Watchdog's signal is highly prioritised, to its absence signifies that the software is non-responsive, and a hard reset is required.
- *Local power supplies* generate the voltages required onboard from the input voltage to support the module boards, SOM, and PCIe devices.
- *PSU monitoring* is based on local voltages that are measured and can be read back by software. This is used for diagnostic purposes and is useful during the production of GWEN. When voltages are out of range, GWEN will stay in reset to prevent faulty or unexpected behaviour.

- *Expansion connector* is implemented for future use. Interfaces like RS232, RS485, USB, I2C, SPI, SPDIF, I2S, etc., can be made available at this connector.
- *JTAG/Debug* interface was used for debugging purposes during the development of GWEN, and for test purposes during production.
- *Status LEDs* are used to indicate the status of GWEN (power, Ethernet, etc.).

3.7.5. Edge Node Firmware

As shown in Figure 14, the firmware of GWEN consists of an operating system (OS) built on top of a hardware abstraction layer (HAL), a container runtime, and additional pre-installed software needed to support enablers. This pre-installed software operates on top of the OS, next to the container runtime, so custom containers can use it. The key specifications of GWEN firmware are:

- *Operating System*. Yocto, based on Linux, is used as the operating system. The Yocto Project is an open source collaboration project that helps developers creating custom Linux-based systems, regardless of the hardware architecture.
- *Hardware abstraction layer (HAL)*. The HAL consists of a device driver as an interface between the electronics and the OS.
- *Configuration and initialisation*. The configuration and initialisation of the standard interfaces (Ethernet, Serial, etc.), SSH, and a default user are preconfigured on GWEN.
- *Container runtime*. Currently, for the container runtime, Docker is used.

At the time of writing, GWEN nodes have been successfully produced, configured, tested, and run in the pilot environment (see Figure 18).

Figure 18. GWEN inside enclosure (bottom left) in a construction site pilot.

Aside from the drivers added to the kernel and modifications to the original setup of the commercial hardware, Docker was installed, combined with applications that support the hardware. This keeps the Linux distribution on the platform lean and clean, and assures one that applications can actually use the available processing power.

3.8. Introducing Selected Enablers

As elaborated, aside from the custom hardware node, the ASSIST-IoT RA has been realised using enablers. Those were of varying type and nature (see Section 2). However, together they formed the global deployment structure of the NGIoT framework, supported by the ASSIST-IoT RA. In addition, custom enablers (other than those specified in the project) can be built and run based on the guidelines, instructions, and created templates

(see Section 3.4). Thanks to the modularity of the architecture, "technically", enablers can be deployed in (almost) any combination, allowing for deployments of varying complexity.

However, to support basic functionalities that an NGIoT deployment must have, and to preserve the principles of enabler design (see Section 3), some enablers are necessary, and others are not. In other words, some elements of the RA are vital in any deployment, while others are complementary. Therefore, a group of *essential* enablers has been delivered by ASSIST-IoT. In this context, the following sections contain brief description of the *essential enablers* and of a group of enablers that are particularly interesting from the NGIoT perspective, as they support autonomic computing. The full list of enablers and their documentation can be found in the official Readthedocs pages of each enabler (https://assist-iot-enablers-documentation.readthedocs.io/en/latest/index.html, accessed on 1 December 2022) and in the ASSIST-IoT project's deliverables. In addition, latest versions of available enablers are published on the project's public GitLab repository (https://gitlab.assist-iot.eu/enablers-registry/public, accessed on 1 December 2022).

3.8.1. Essential Enablers

Essential enablers deliver basic functionalities that are expected in an NGIoT deployment. Hence, they should be tackled first, whenever undertaking an installation an NGIoT ecosystem. For the essential enablers, the following aspects were considered:

- *Data distribution*: the principal aim of an IoT deployment is to move data/information (status, action orders, etc.) across the system (sensors, edge nodes, cloud, etc.).
- *Data storage*: to exploit such data, there must be (at some point of the computing chain) a place where data will reside.
- *User interface*: to manage devices, applications (i.e., enablers), users, etc.
- *Security*: of applications, data flow and system access (authentication, authorisation).
- *Location*: devices, nodes, users, and enablers might become part of the ecosystem from distant locations, both geographical and topological (i.e., outside of the local network).

After careful analysis of the NGIoT requirements and the ASSIST-IoT RA, 11 out of 41 ASSIST-IoT enablers were identified as essential. This should help administrators decide on the order of enabler deployment. However, all enablers installed via the predefined ASSIST-IoT installation scripts are "essential". This means that, for all practical purposes, some enablers can be considered as "preinstalled". Naturally, other enablers can be deployed immediately afterwards, e.g., with the help of the Smart Orchestrator (see Section 3.5). Table 1 includes a list of the essential enablers of the ASSIST-IoT architecture, and the summary of the reasons why those have been selected.

Table 1. ASSIST-IoT essential enablers summary.

Enabler	Rationale	Pre-Installed
Smart Orchestrator	The "brain" and "engine" of the deployment, allowing the installation of the remaining enablers, via the kubectl proxy tool (see Section 3.5).	yes
VPN enabler	Enabler in charge of facilitating access to a node, or device, from a different network to the site's private network using a public network (e.g., the Internet), or a non-trusted private one.	
Edge Data Broker	Facilitator of data moving across enablers in the deployment. Particularly relevant where data from IoT devices must be distributed.	
LTSE	Designed to facilitate storage requested by other enablers in the deployment; moreover, to keep enablers status information for potential recovery, other roles, and to store access related information.	

Table 1. *Cont.*

Enabler	Rationale	Pre-Installed
OpenAPI Manager	Allows external parties (not owners of the deployment) to access enablers' data and functionalities.	
Tactile Dashboard	Centralised user interface for configuration of the deployment. Main interface and access gate for humans to the deployment.	yes
Security enablers	Must-have security features, such as identity management, authentication, authorisation and access and policy rules specification.	
Manageability enablers	Allow interaction with the orchestrator, oversee enablers and nodes status, allow combination of enablers to deliver data transfer within the deployment.	yes

3.8.2. Self-* Enablers

Among the ASSIST-IoT enablers, those that follow the autonomic computing conventions (called "self-*") are of particular interest for the NGIoT use-cases. Even though use of AI/ML is widespread in IoT, autonomic computing is rarely an explicit part of IoT architectures. Although the self-* enablers do not belong to the "essentials", collectively they offer an important set of tools to support semi-autonomous operations for IoT deployments. In particular, they provide mechanisms for: self-healing, resource provisioning, location tracking/processing, and automated configuration. All except the first one (see next) offer fully encapsulated deployments, ready to be used in a k8s-powered environments.

The *self-healing device enabler* aims at providing IoT devices with capabilities to actively perform recovery procedures, in cases of encountering abnormal states. The "failure conditions" handled by the enabler can be divided into three categories: security, dependability (e.g., data corruption or network protocol violation), and long-term (hardware's end-of-life, or unsupported hardware capabilities), based on a pre-established schedule of routines. Logically, the enabler has been divided into three components: a self-detector, a self-monitor, and a self-remediator. The goal of the first component is to collect diagnostic information, such as memory/CPU usage, memory access, and network metrics (RSSI levels), from the IoT device. Subsequently, the self-monitor analyses the collected data, and on that basis, assesses state of health of the device. If, based on a predefined set of rules, the result indicates malfunctioning or intrusion, the third component—the self-remediator—attempts a remediation action. If after the remediation, the device is not back to the normal state, the component is self-triggered to select another remediation process from the list. Due to its low-level, and to host operating system environment access requirements, the self-healing device enabler is the only non-containerised solution within the self-* group.

The main task of the *resource provisioning* enabler is to dynamically adapt the autoscaling of nodes and clusters, optimising the resource utilisation and general operation of IoT deployments. In particular, its role is essential for edge deployments, where resources are much more constrained than in the cloud, and where static resource allocation techniques become unfeasible. In its operation, the resource provisioning enabler applies ML techniques to predict the future resource consumption requirements, based on historical data. It is also capable of supervising multiple deployments in a cluster, and analysing and controlling the resource utilisation for each of them, independently.

The *location tracking* and the *location processing* are enablers that allow IoT deployments to utilise geolocation and geofencing. The location tracking enabler is a low-level, hardware-related solution that allows one to receive the positioning data from geofencing tags. Internally, the enabler consists of two main components: a "tag & anchor configuration" component, and a "localisation engine" component. The former is used to set up working parameters for the anchors, whereas the latter is responsible for transforming low-level data

into the proper geofencing/geolocation representation. The location processing enabler, on the other hand, provides highly configurable and flexible geofencing capabilities, based on the (geo)location data. It handles data updates and queries, using both the HTTP-based request-response and the streaming approach.

The *automated configuration* enabler aims at keeping heterogeneous devices and services synchronised with their configurations. For each of the controlled artefacts, it allows one to define a flexible, parameterised configuration structure, including alternatives that can be considered/applied in case of errors. Based on the monitoring data coming from the clients, the enabler will detect if a fallback configuration should be used and will apply it in reaction to changes in the environment as necessary/required. To adequately react to errors and other events/conditions, the enabler utilises a flexible representation of the configuration and intelligent mechanisms that, based on user-defined rules, can automatically decide which functionalities will be kept in the event of limited available resources.

4. Pilot Deployments

The described enablers, and others implemented in the scope of the ASSIST-IoT project, offer independent functions and can be used as-is in any application domain. They can be combined to form specific deployments. From the beginning of the project, it was a priority to validate them in real-world scenarios. ASSIST-IoT RA, the developed software, and the GWEN hardware, are being used in pilot deployments, spread across three highly heterogeneous IoT domains. At the time of writing the pilot, trials are ongoing, with promising, intermediate results [35].

The first pilot ("Pilot 1: Port automation") operates in the domain of maritime port automation and optimisation of container handling. Timing of operations in a port is critical and requires high reliability and resilience, given the high volume of containers that pass through ports daily. The containers need to be loaded, unloaded, and stored, with complex transport operations in-between those stages. Additionally, a port environment includes many internal and external stakeholders (transport companies, terminal operators, ship operators, port authorities, etc.) and a variety of heavy equipment (cranes, gantries, trucks, and the containers themselves). The busy nature of a port combined with large open areas is a challenge when it comes to networking, and as a consequence, so are quick identification and management of containers. Given the fact that all those operations may need to be done remotely, it is a challenging environment.

ASSIST-IoT aims to solve or improve existing solutions for a number of port operation problems. These include traceability of containers, automation of crane operation, and remote control of gantries (with support for AR interface). To this end, GWEN nodes are being deployed as part of a local hardware environment (spread across the large area of the port). At different points, in this environment, enablers for network management and reliability, geolocation, self-configuration, and AR augmentation (among others) are deployed.

The second ASSIST-IoT pilot ("Pilot 2: Smart Safety of workers") aims at improving safety of workers in a construction site. At various stages of development, a construction site can be a very hazardous environment, with dynamically changing conditions, high level of noise, heavy machinery, chemical fumes, and other dangers. As for that, there are a number of regulations and restrictions that try to preserve the health (and possibly even lives) of workers. It is very common for such an environment to include heavy machines with low visibility on the side of the operator, which are put onto and removed from the site at various stages of the construction. Additionally, personnel that needs to work on the site includes not only skilled, experienced, and safety-certified physical workers, but also seasonal workers with less experience, inspectors, and managers that may need to visit the site.

Improvements in safety conditions that ASSIST-IoT bring to this pilot are centred around monitoring the dynamic conditions of the site and workers. It includes gathering and processing of individual workers health data (e.g., temperature), position, and usage of equipment. These data are overlaid on a 3D map of the site, where dangerous and

privileged areas are marked and updated. Through the use of personal cooling systems (vests), accelerometers and thermometers (in smart watches), cameras, and location tags and anchors (controlled via GWENs), the features of health monitoring, fall detection, safety equipment compliance, and unsupervised or prohibited entry detection are delivered. Geolocation, federated learning, semantic data annotation and translation, and edge data broker enablers (among others) play crucial roles here.

The third pilot ("Pilot 3: Cohesive vehicle monitoring and diagnostics") covers two distinct groups of use-cases within vehicle fleet management and monitoring. The first one exploits the data about operation of a modern car, where sensors deliver detailed information about the status and statistics of the operation of individual subsystems and the whole vehicle. These data are gathered, analysed, and processed partially on edge nodes, and in part in the cloud. The results of the analysis are used to create a feedback loop and tune the parameters of the powertrain operation to optimise, and in effect decrease, vehicle fleet emissions. Data from cars are gathered and transported through ASSIST-IoT network and data management plane enablers, and processed with the AI/ML enablers.

The second part of the vehicle-monitoring pilot explores a different direction. It focuses on visual inspection of the condition of cars, in the context of maintenance service of rental, or leased, vehicles. Here, any changes in external condition of vehicles may be grounds for insurance claims, so it is central to the operations of the stakeholder (i.e., fleet owner) to systematically identify, and document, the state of the car. This task brings challenges related to large amount of data, in form of car pictures, with high variability of conditions such as cleanliness of the body, camera angles, and weather conditions, when taking pictures. Some problems can be solved by ensuring that the data gathering environment is constant. To this end, a special gate, bespeckled with fixed cameras and lights, was designed (outside of the ASSIST-IoT context). Other problems are related to very large amounts of data generated per car, which needs to be processed with accuracy high enough to warrant acceptance or rejection of insurance claims (in case of any dents or more serious damages). Here, ASSIST-IoT aims to decrease the amount of data that need to be sent over the network by processing some of it in edge nodes. The federated learning enablers are crucial for this pilot, as they deliver both edge processing capabilities and the AI needed to automatically label large images in ample quantities.

Pilot Deployments—Discussion

First, let us note that the pilot domains cover very different domains. Therefore, while the set of problems to be solved may be overlapping, they also necessarily contain issues that are quite unique. In this context, the key point is that the same enablers can be (and are) used in different pilots to address similar but not identical questions. For example, the Edge Data Broker Enabler (EDBE) is a general data streaming solution designed to route the data. Moreover, it can filter its content. Multiple instances of this enabler can be deployed at different points in the architecture. In the construction site pilot this enabler is used to route data between devices (vests, watches, etc.) and dashboards (e.g., Tactile Dashboard Enabler) and/or data storage. It also generates events and sends them to proper recipients. In the maritime port pilot, there are also multiple instances of EDBE, each dedicated to a single "trial". These instances had less "individual responsibility" (e.g., they support simple transport of data to and from a single crane and its operator). In the vehicle monitoring pilot, EDBE is replicated to support large amount of data, originating from a large number of vehicles in a fleet. In this particular case, EDBE supports a relatively simple data transport scheme.

Another noteworthy enabler is the Long-Term Storage Enabler (LTSE), which provides a secure, scalable, and resilient storage solution for ASSIST-IoT RA based applications. It offers a general-purpose SQL and NoSQL database space suitable for permanent storage of data. The need for data archiving, or even logging data temporarily (e.g., for a week), is ubiquitous in IoT. Unsurprisingly, it comes up in *all* ASSIST-IoT pilots as well. For example, personal data of workers, in Pilot 2, are persisted (possibly indefinitely) in a

single, replicated instance of LTSE. In pilot 3, the pictures of cars are stored by individual instances of LTSE, to facilitate the temporary (i.e., a couple of hours/days) storage needs of the federated learning enablers. Data are persisted locally, within the privacy scope of any given FL node. Similarly, data concerning performance of individual cars or movement of port containers are persisted using LTSE instances.

A very important example of project developed artefacts is the gateway edge node (GWEN). It is widely used within the pilots. Here, GWEN acts as a host for enablers and can be used as a general-purpose node or a platform for specific hardware modules. Here, depending on the pilot, GWEN can be set-up to facilitate specific functions. For instance, location tracking and location processing enablers are present in port and construction site pilots. They are used to track machinery and equipment (e.g., containers) in the port pilot and workers in the construction site safety pilot. To do so, they will be deployed within pertinent GWEN instances. In this way, they can be deployed and used in any other IoT deployment, where location tracking is needed. However, when they are not needed (e.g., the automotive pilots), they will not be deployed, thereby reducing resource consumption within the GWEN.

The pilots also differ when it comes to the data processing needs. A semantic annotator enabler is used in the construction pilot to annotate data coming from heterogeneous source devices (vests, smartwatches, location tags, etc.) delivered by different vendors, with different data types and standards. Semantic annotation is useful in such cases to facilitate data (stream) interoperability and to allow consistent processing. Therefore, in this and other similar situations, when data interoperability needs to be facilitated, the semantic annotator enabler can be deployed (e.g., on the GWEN node). On the other hand, in the port pilot, all data need to pass through a port management system that is already integrated with all relevant systems. Moreover, integration with potential future partners is constrained by contractual arrangements, typical to the port ecosystems. For these (business) reasons, a semantic annotator enabler will not be deployed in this context.

As can be seen, by virtue of modularity and virtualisation, ASSIST-IoT enablers can be picked and chosen to fit many use-cases. With the exception of hardware elements (that need to be physically installed), the process of deployment and management of enablers is homogenised and standardised. Moreover, in this way, they are well designed to be used as close to the edge as possible (and needed). Specifically, the need for enabler deployment close to the edge can guide the setup of appropriate GWEN nodes.

5. Concluding Remarks

The ASSIST-IoT RA delivers a multi-layered solution for NGIoT deployments, covering a wide range of needs and use-cases. The architecture divides complex systems, within the area of NGIoT, into four planes and five verticals, each grouping different functions. It is a highly modular, customisable, scalable, and flexible approach that fits the dynamic nature of NGIoT.

A core idea within ASSIST-IoT RA is the concept of *encapsulated enablers*—packaged and orchestrable micro-applications, enabling preservation of the security and coherence of complex systems, comprised of both software and hardware components, while allowing for their customised and dynamically changing deployments. On top of the design documents and theory, ASSIST-IoT delivers a comprehensive set of implemented enablers, together with a state-of-the-art, modularised runtime environment, which exhibits all the features promised by the architecture.

Additionally, custom hardware nodes were designed and produced to support edge applications and provide a foundation on which the enablers can run. The hardware is designed to follow the familiar architectural principles of flexibility and adaptability. It is a modular system with swappable components that communicate through standardised interfaces and can scalably answer the needs of most IoT use-cases. Naturally, ASSIST-IoT RA is also compatible with other hardware implementations of edge nodes. The relatively relaxed firmware requirements allow for a wide array of devices to be used, and the GWEN

implementation is just an example of how the architectural principles are realised on the hardware level.

ASSIST-IoT comes with first-class support of 5G and inclusion of the Tactile Internet concept in the design. The Smart Network and Control Plane is prepared to handle rapid increases in traffic and provides tools to scale network functions. This allows systems to meet the desired QoS in face of NGIoT network requirements.

In contrast with other existing RAs, ASSIST-IoT RA delivers not only theoretical principles, but also executes them by delivering (reference) implementation of software components and physically ready hardware, and by applying them to real world use-cases.

Through the concept of enablers, and the general focus on adaptability, scalability, and modularity (expressed also in the hardware), ASSIST-IoT implementations are less restrictive than those based on other RAs. Consequently, whereas previously it was not uncommon for an RA to be introduced without trial implementations, or with (an) implementation(s) that worked in (few) selected domain(s) and was (were) extremely difficult to transfer to other domains, or use-cases, the presented approach is highly "repeatable" and "transferable". As discussed, essential enablers provide core functions for IoT deployment in any domain. Moreover, all implemented enablers were validated to assure that they are domain agnostic. Finally, thanks to the implemented management tools, enablers can be picked-and-chosen as needed.

Hence, possibly for the first time in the Internet of Things domain, a "complete solution" has been proposed. It is domain agnostic and consists of a reference architecture and a comprehensive set of validated enablers that allow it to be used to instantiate NGIoT-compliant ecosystems.

Author Contributions: This contribution is a result of a team effort, supported in part by external funding. While the lead author formed the foundational ideas, all authors participated equally in all remaining aspects of the project. All authors have read and agreed to the published version of the manuscript.

Funding: This work was funded by the European Commission, under the Horizon Europe project ASSIST-IoT, grant number 957258.

Data Availability Statement: All existing and future (public) results of the ASSIST-IoT project are and will be made available at https://assist-iot.eu/.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

RA	Reference Architecture
NGIoT	Next Generation Internet of Things
SDN	Software defined networking
NFV	Network function virtualisation
VR	virtual reality
AR	augumented reality
DLT	distributed ledger technologies
RIA	research and innovation actions
GWEN	Gateway Edge Node
EDBE	Edge Data Broker Enabler
API	Application Programming Interface
BIM	Building Information Modelling
CA	Certificate Authority
CHE	Container Handling Equipment

CNI	Container Network Interface
CPU	Central Processing Unit
DLT	Distributed Ledger Technology
ES	Enabler Stories
FL	Federated Learning
gRPC	gRPC Remote Procedure Calls
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
IdM	Identity Manager
IoT	Internet of Things
JSON	JavaScript Object Notation
k8s	Kubernetes
ML	Machine Learning
MQTT	MQ Telemetry Transport
NoSQL	Not Only SQL
OS	Operating System
PAP	Policy Administration Point
PDP	Policy Decision Point
PEP	Policy Enforcement Point
PIP	Policy Information Point
REST	Representational State Transfer
SD-WAN	Software-Defined Wide Area Network
SQL	Structured Query Language
RAM	Random Access Memory
RTG	Rubber-Tyred Gantry (crane)
UV	Ultraviolet
VPN	Virtual Private Network
VIM	Virtualised Infrastructure Manager
k8s	Kubernetes
UWB	ultra wide band

References

1. D3.1. IoT research, innovation and deployment priorities in the EU. In *NG-IoT Project*; 2020. Available online: https://www.ngiot.eu/wp-content/uploads/sites/73/2020/09/D3.1.pdf (accessed on 1 December 2022).
2. A Vision of the Future Internet. In *NGI Project*; 2020. Available online: https://www.ngiot.eu/ (accessed on 1 December 2022).
3. Nguyen-An, H.; Silverston, T.; Yamazaki, T.; Miyoshi, T. IoT traffic: Modeling and measurement experiments. *IoT* **2021**, *2*, 140–162. [CrossRef]
4. International Data Corporation: Worldwide Global DataSphere IoT Device and Data Forecast, 2019–2023. Available online: https://www.iotplaybook.com/tags/worldwide-global-datasphere-iot-device-and-data-forecast-2019-2023 (accessed on 1 February 2023).
5. Internet of Things (IoT) and Non-IoT Active Device Connections Worldwide from 2010 to 2025. Available online: https://www.statista.com/statistics/1101442/iot-number-of-connected-devices-worldwide/ (accessed on 1 February 2023).
6. Fanibhare, V.; Sarkar, N.I.; Al-Anbuky, A. A survey of the tactile internet: Design issues and challenges, applications, and future directions. *Electronics* **2021**, *10*, 2171. [CrossRef]
7. Baumgartner, M.; Juhár, J.; Papaj, J. Short performance analysis of the LTE and 5G access technologies in NS-3. In Proceedings of the 2021 16th Conference on Computer Science and Intelligence Systems (FedCSIS), Online, 2–5 September 2021; pp. 337–340.
8. Gupta, P.; Iyer, S. *Goodbye, Motherboard. Hello, Silicon-Interconnect Fabric*; IEEE Spectrum: New York, NY, USA, 2019.
9. Conti, F.; Schilling, R.; Schiavone, P.D.; Pullini, A.; Rossi, D.; Gürkaynak, F.K.; Muehlberghuber, M.; Gautschi, M.; Loi, I.; Haugou, G.; et al. An IoT endpoint system-on-chip for secure and energy-efficient near-sensor analytics. *IEEE Trans. Circuits Syst. I Regul. Pap.* **2017**, *64*, 2481–2494. [CrossRef]
10. Liu, X.; Sánchez-Sinencio, E. A highly efficient ultralow photovoltaic power harvesting system with MPPT for internet of things smart nodes. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **2015**, *23*, 3065–3075. [CrossRef]
11. Urbina, M.; Acosta, T.; Lázaro, J.; Astarloa, A.; Bidarte, U. Smart sensor: SoC architecture for the Industrial Internet of Things. *IEEE Internet Things J.* **2019**, *6*, 6567–6577. [CrossRef]
12. Ali, O.; Ishak, M.K.; Bhatti, M.K.L.; Khan, I.; Kim, K.I. A Comprehensive Review of Internet of Things: Technology Stack, Middlewares, and Fog/Edge Computing Interface. *Sensors* **2022**, *22*, 995. . [CrossRef] [PubMed]
13. Jebbar, Y.; Promwongsa, N.; Belqasmi, F.; Glitho, R.H. A Case Study on the Deployment of a Tactile Internet Application in a Hybrid Cloud, Edge, and Mobile Ad Hoc Cloud Environment. *IEEE Syst. J.* **2021**, *16*, 1182–1193. [CrossRef]
14. Fazeldehkordi, E.; Grønli, T.M. A Survey of Security Architectures for Edge Computing-Based IoT. *IoT* **2022**, *3*, 332–365. [CrossRef]

15. Sharma, S.K.; Woungang, I.; Anpalagan, A.; Chatzinotas, S. Toward tactile internet in beyond 5G era: Recent advances, current issues, and future directions. *IEEE Access* **2020**, *8*, 56948–56991. [CrossRef]
16. Dogra, A.; Jha, R.K.; Jain, S. A survey on beyond 5G network with the advent of 6G: Architecture and emerging technologies. *IEEE Access* **2020**, *9*, 67512–67547. [CrossRef]
17. Asheralieva, A.; Niyato, D. Optimizing Age of Information and Security of the Next-Generation Internet of Everything Systems. *IEEE Internet Things J.* **2022**, *9*, 20331–20351. [CrossRef]
18. Amin, F.; Abbasi, R.; Mateen, A.; Ali Abid, M.; Khan, S. A Step toward Next-Generation Advancements in the Internet of Things Technologies. *Sensors* **2022**, *22*, 8072. [CrossRef] [PubMed]
19. nexT gEneRation sMart INterconnectEd ioT. *D1.3 Project Scientific/Technical Plan*; nexT gEneRation sMart INterconnectEd ioT: Luxembourg, 2021.
20. IoT-NGIN. *D1.2 IoT Meta-Architecture, Components, and Benchmarking*; IoT-NGIN: Issy-les-Moulineaux, France, 2021.
21. Trakadas, P.; Masip-Bruin, X.; Facca, F.M.; Spantideas, S.T.; Giannopoulos, A.E.; Kapsalis, N.C.; Martins, R.; Bosani, E.; Ramon, J.; Prats, R.G.; et al. A Reference Architecture for Cloud–Edge Meta-Operating Systems Enabling Cross-Domain, Data-Intensive, ML-Assisted Applications: Architectural Overview and Key Concepts. *Sensors* **2022**, *22*, 9003. [CrossRef] [PubMed]
22. Fornés-Leal, A.; Lacalle, I.; Palau, C.E.; Szmeja, P.; Ganzha, M.; Paprzycki, M.; Garro, E.; Blanquer, F. Assist-iot: A reference architecture for next generation internet of things. In *New Trends in Intelligent Software Methodologies, Tools and Techniques*; IOS Press: Amsterdam, The Netherlands, 2022; pp. 109–128.
23. Júnior, A.A.; Misra, S.; Soares, M.S. A systematic mapping study on software architectures description based on ISO/IEC/IEEE 42010: 2011. In Proceedings of the International Conference on Computational Science and Its Applications, Saint Petersburg, Russia, 1–4 July 2019; Springer: Berlin/Heidelberg, Germany, 2019; pp. 17–30.
24. Chaabane, M.; Bouassida, I.; Jmaiel, M. System of systems software architecture description using the ISO/IEC/IEEE 42010 standard. In Proceedings of the Symposium on Applied Computing, Marrakech, Morocco, 4–6 April 2017; pp. 1793–1798.
25. Rozanski, N.; Woods, E. *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives*; Addison-Wesley: Boston, MA, USA, 2011.
26. IOT-A Project Consortium. *Final Architectural Reference Model for the Iot*; Technical Report; VDI/VDE Innovation+ Technik GmbH: Berlin, Germany, 2013.
27. ITU-T. *Y.2060: Overview of the Internet of Things*; ITU-T: Geneva, Switzerland, 2012.
28. WSO2. *A Reference Architecture for the Internet of Things*; WSO2: Santa Clara, CA, USA, 2015.
29. OpenFog Consortium Architecture Working Group OpenFog reference architecture for fog computing. *OPFRA001* **2017**, *20817*, 162.
30. ECC. *All Edge Computing Reference Architecture 2.0*; ECC: Champaign, IL, USA, 2017.
31. Standardisation, AIOTI WG03–IoT. *High Level Architecture (HLA) Release 5.0*; Internet of Things Innovation (AIOTI): Brussels, Belgium, 2020.
32. IC. *The Industrial Internet of Things Volume G1: Reference Architecture v1.9*; IC: Jacksonville, IL, USA, 2019.
33. VDI/VDE Society Measurement and Automatic Control (GMA). *Status Report Reference Architecture Model Industrie 4.0 (RAMI4.0)*; GMA: New York, NY, USA, 2015.
34. Project FE. *D2.4. FAR-EDGE Architecture and Components Specification*; Project FE: St. Boulder, CO, USA, 2017.
35. Project AI. *D7.3 Pilot Scenario Implementation–Intermediate Version*; Project AI: Perth, Australia, 2022.
36. Kruchten, P.B. The 4+ 1 view model of architecture. *IEEE Softw.* **1995**, *12*, 42–50. [CrossRef]
37. Project AI. *D3.7. Architecture Definition–Final*; Project AI: Perth, Australia, 2022.
38. ETSI. *GS NFV-MAN 001 Network Functions Virtualisation (NFV); Management and Orchestration*; ETSI: Sophia Antipolis, France, 2014.
39. Project AI. *D2.4. Ethics and Privacy Protection Manual*; Project AI: Perth, Australia, 2022.
40. Project AI. *D6.1 Devsecops Methodology Additionally, Tools*; Project AI: Perth, Australia, 2022.
41. Kumar, R.; Goyal, R. Modeling continuous security: A conceptual model for automated DevSecOps using open-source software over cloud (ADOC). *Comput. Secur.* **2020**, *97*, 101967. [CrossRef]
42. 5G-PPP Software Network Working Group. *Cloud-Native and Verticals' Services*; 5G-PPP Software Network Working Group: Heidelberg, Germany, 2019.